For more information, contact:
Nanette Collins
Public Relations for Verific Design Automation
(617) 437-1822
nanette@nvc.com

Using Verific's System Verilog and VHDL processing tools

Andy Fox
The Really Useful Software and Hardware Company (RUSHC)
Bonny Doon, Calif.

and
Hovhannes Ter-Milqsetyan
Tigran Sargsyan
Vigen Gasparyan,
RUSHC R&D LLC,
Yerevan, Armenia
www.rushc.com

Verific Design Automation is known for its Verilog, VHDL and SystemVerilog language processing sub-systems. Since its founding in 1999, its software has served as the front end to a wide range of electronic design automation (EDA) and field programmable gate array (FPGA) tools for analysis, simulation, verification, synthesis, emulation and test of register transfer level (RTL) designs.

This article offers a look at how a design team would use Verific's tools to ensure the success of an EDA project. It outlines what the Verific tool kit does, its internal representations and strategies for success, along with a few notes of caution. Finally, the article describes "Veriapps", a package of utilities for Verific developers.

Verific Toolkit Capabilities

Verific's tool kit reads in hardware description languages (HDLs) — SystemVerilog, Verilog and VHDL. Depending on the amount of elaboration needed, it offers four levels of abstraction:

- The parse tree

- The statically elaborated parse tree
- The operator netlist
- A gate level netlist

The parse tree representation is generated during the analysis phase and traversed using a walker. Verific provides a template for walkers so each syntactic category can be easily traversed. Static elaboration further expands the parse tree by resolving parameters, generating statements and other statically determinable aspects of the HDL.

For example, if a developer wants to add a type checker to catch a mismatch in sizes in a VHDL relational operator, it could be done using the VhdlExpression walker class. The Verific code elaborates it in a way that the developer can get a type for any identifier reference. Each identifier instance is represented as an IdReference which in turn refers to an IdDefinition field from which all type information about the identifier can be extracted..

The parse tree can be evaluated to further resolve data types, as necessary — such as, when evaluating recursive functions or resolving dynamically assigned ranges in VHDL. Mixed-language support is provided using the vl_types.vhd package that permits various type conversions.

Full elaboration translates the parse tree to a netlist by performing a synthesis step. With a little care, operators can be preserved and a bus-oriented or unflattened data model extracted, called the "operator netlist."

This is a useful level of abstraction because it provides the semantics of the language using operators, including decoders, shifters and state elements. It also has a "wide operators" function on bus level structures, known as netbus or portbus in Verific terminology.

The same netlist view is generated for all languages, allowing a high degree of language independence. This would be a good starting point, for example, if the application involves writing a simulation accelerator for synthesizable designs written in any HDL supported by Verific. Moreover, because Verific has taken care of the language processing, the developer only needs to understand the netlist.

The Verific netlist follows a traditional Electronic Design Interchange Format (EDIF)-type hierarchy of library, cells, instance, view and ports. This model is efficient and familiar to most EDA developers.

Using the flattening application programming interface (API), the Verific toolkit will flatten the operator-level netlist to primitive gates, such as PRIM_AND, PRIM_OR and PRIM_XOR. At this level, the full logic of the design is available.

Verific Data Structures	**Abstraction Level**	**Application Domain**
VeriModule*, VeriVisitor*	Parse Tree/statically elaborated Parse Tree.	Language-specific operations; Language checking, HDL extraction.
Instance* Types: Arithmetic operators (OPER_ADDER …), Memories (OPER_READ_PORT, OPER_WRITE_PORT), Wide logical operators (OPER_WIDE_AND, OPER_WIDE_OR)	Operator Netlist	Synthesis-specific netlist transformations, including operator selection and RTL interpretation viewing.
	Primitive Netlist	Gate-level design analysis, such as technology mapping and gate-level equivalency checking.

Figure 1 Caption: *The levels of abstraction support by Verific's tool kit and their applicability are illustrated in this diagram*.

Integration of Verific into the Software Flow

Verific's C++ APIs can be called directly from a developer's code, after correctly linking in the Verific libraries. The APIs most often used by developers are Analyze and Elaborate.

Entry to the Verific data structures is usually through the top-level parse tree module (Verific API: veri_file::GetTopModules or netlist (Netlist::PresentDesign()).

A side note: A wrapper to the Verific APIs is needed to provide a "Design Manager" that harvests all appropriate HDL files and resolves library issues discussed later.

At this point, most developers translate Verific structures to their own database and continue independently of the Verific code base. However, the Verific netlist is sufficiently powerful and stable to justify its use in the developer's environment. The issue remains how to do this in a way that protects the developer's code base investment while safely exploiting the Verific code.

RUSHC has experimented with three methods:

(a) Deriving the Verific classes, including Instance and Cell, from a generic abstract class.
(b) Deriving customer-specific classes from the Verific classes.
(c) Templatization of code. Developing algorithms parameterized by abstract types and providing concrete interfaces that use Verific "hardened" types.

While (a) and (b) effectively allow all Verific APIs to be accessible natively from customer code, (c) offers the most generality by quickly allowing core algorithms to be ported to new structures and keeps the style adopted by the boost libraries. To demonstrate, consider the representation of a "cut" in a netlist, where a cut is nothing more than a group of pins delineating a region of a netlist. The abstract class for a cut is outlined below.

```
template <typename DesignObjectType>
class CutAbstract
    : private Rushc::Base::UnCopyAble
{
public:
    CutAbstract();
// . . .

template <typename Predicate>
    typename std::vector<DesignObjectType>::const_iterator
FindDriver(Predicate) const;

    const std::vector<DesignObjectType>& gOps() const;

    const std::vector<DesignObjectType>& gDrivers() const;

    virtual bool operator == (const CutAbstract&) const;

    virtual bool operator != (const CutAbstract&) const;


protected:
    std::vector<DesignObjectType> m_drivers;

    std::vector<DesignObjectType> m_outputs;

    typedef typename boost::unordered_set<DesignObjectType> DriversSet;

};..
```

Note that the DesignObjectType is left as a type parameter. The concrete version of the cut and its various algorithms is then realized by “hardening” the type of the DesignObjectType to the Verific specific type “PortRef”:

```
class Cut
    : public Rushc::Base::CutAbstract<const Verific::PortRef*>
{
public:
    Cut();

    virtual ~Cut();

    bool IsDegenerateCut() const;

    void Print() const;

}; // class Cut
```

Using this approach, RUSHC built a set of library routines and useful applications for formal verification and synthesis that can exploit the full power of the Verific database while offering generality for use on other structures. For example, RUSHC ported a full formal verification method approach for generating timing exceptions from Verific to the customer database in days.

Case Studies, Strategies for Success, Notes of Caution

In our experience using the Verific tool kit, the key to success is deciding on the most appropriate level of abstraction in the Verific flow, whether it's a parse tree, operator netlist or primitive netlist. A key point is deciding where the developer wants to innovate. Problems will arise if this decision is not carefully thought through. For instance, given that Verific provides a generic "operator" netlist view applicable to all languages, a developer would need a compelling reason to rewrite the elaborator part of Verific for his or her application.

The following case samples can help set the context for such decisions.

Company A: A Mapping Solution

The project required the development of a high-speed technology mapping solution. Operators needed to be extracted by library operators and novel cut generation and matching algorithms devised.

The Verific primitive netlist level of abstraction was selected, freeing the EDA developers from having to code HDL processing/elaboration and instead focus on the core new algorithms. The Verific netlist produced by elaboration was further improved by Sat-Sweeping [6], A templatized mapping and matching solution was devised which could be run both on the Verific netlist and the data structures used post placement and routing. . The innovation was correctly focused on where the EDA developer added most value. The lessons included the value of templatization and the stability and efficacy of the Verific database.

Company B: RTL Acceleration for a Synthesizable HDL Subset

The project required the extraction of novel instructions, including MULT, LD, ADD, STR with hardware specific ones eg "TAP x [10:0]" to fish bits out of busses, from an HDL for execution on a simulation hardware accelerator. The company had initially decided to "roll its own" elaborator and was investing expensive resources in resolving complicated HDL issues – including many of those those outlined in Sutherland's "Standard Gotchas Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know" [2].

However, the company only needed the synthesizable HDL subset. After some coaxing of the client, the operator level netlist from Verific was used as the basis for the machine instruction generation and, within a few weeks, instructions were being generated. The focus of the project shifted to "how to establish the value of high-speed simulation acceleration" from "how to resolve complicated HDL issues."

The lesson learned by this company is to avoid becoming an HDL processing guru unless that is the core business.

Company C: Formal Methods for Timing Exception Checks

Another project required developing and applying timing exception proving algorithms using formal methods such as the And Invert Graph (AIG) techniques.

From the start, data structures beyond the Verific netlist would need to be used and Verilog counter examples would need to be generated and, where possible, references to the users' source HDL made. For example:

```
sequence LSA_3__CD_3__PATH_IS_NOT_SENSITIZED_1_CYCLES;
      (! pulse);
```

```
endsequence
```

The Verific netlist database was chosen as the starting point, freeing the EDA team from learning HDLs. A utility for translating the Verific netlist to an ABC [7] AIG was devised and a method for correlating inputs/outputs and “user nets” of both netlists was constructed. Verific’s code has an API for marking nets that appear in the user’s source HDL as a “user net”, net -> isUser().

High-speed algorithms for simulation, sat clause generation bounded model checking and fixed-point analysis were devised. As before, a templatized approach to algorithm development was used.

A trade-off was made between algorithms best run on the Verific netlist database, such as simulation for candidate filtering, and those best run on the AIG model such as bounded model checking and ternary simulation for fixed point analysis. By starting with the Verific database annotated with user source/line information, algorithms were provided with useful information about the original design. Using the templatization approach, the core algorithms developed were easily applied to both the company’s database and the Verific database.

Wish List of Verific Functionality

In the course of executing multiple Verific projects, we have accumulated a list of items of benefit to design teams. These are generic utilities that could be in the Verific code base but are not and include:

- A package for checking the parse tree for language gotchas and warnings, such as those outlined in [2] and those in the Semiconductor Reuse Standard [5]. Every design team wants better error reporting.
- Redundancy removal, sat-sweeping [6]. Many applications need to start from the smallest netlist possible.

- Design Manager. The Verific APIs provide basic “analyze” and “elaborate” type functionality and support for Verilog-XL / VCS type –f options. Most applications demand a wrapper for supporting command line arguments, handling multiple libraries and invoking VHDL file sorting etc. This is common code that should be shared.
- Core algorithms running on Verific netlist representations for verification, library-based technology mapping, pretty printing, including dot file generation, and basic netlist utilities, such as standard depth first search and breadth first search functors.

RUSHC developed the Veriapps (for Verific Applications) package to fulfill some of these requirements.

Veriapps

The Veriapps library of utilities for the Verific tool chain from RUSHC operates on the Verific data structures. The packages shown in the table below are accessible via C++ or the Verific Perl interface, and the code is available in source form.

Package	**Data Structure Level**	**Description**
Generic walker	Parse Tree	Generic parse tree walker with callback mechanism for custom language checks.
STPGen	Operator Netlist	Generates Simple Theorem Prover (STP)[8] data structures and Saturated Module Theorem (SMT)-2 formats for use with STP and Z-3 [9] solvers.
Abcintf	Primitive Netlist	Source-level integration of the abc package [7] and includes access to all abc commands for synthesis and verification.
Sat-sweep	Primitive Netlist	Network clean up. Sat sweeping, constant removal, common logic extraction to reduce the size of the netlist.
NwkSim	Operator Netlist, Primitive Netlist	Simulation engine is an application of dfs to allow random and constrained random simulation on Verific netlists at the primitive and operator level.
Formal Verify	Primitive Netlist, Operator Netlist	Formal verification using Sat of two Verific netlists for checking transformations.

Cgate	Primitive Netlist	Generation of clock gates for source Verilog. The two cases considered are flop output is unobservable on next cycle, and flop input is proven not to change on next cycle.
Nwk, Util,IO, Cut	Operator Netlist Primitive Netlist	Utilities for traversing Verific netlist, such as BFS and DFS, cut generation, cut representation, i/o including dot-file generation.
Liberty, tech_map	Primitive Netlist	Technology mapper onto Liberty library cells.
Llvm interface	Operator Netlist	Generates operator netlist from llvm ir [4]. Algorithms include SDC scheduling [3], resource assignment and mapping of sequences of instructions to library.

Conclusion

It's often advised that companies and tool developers stick to their core competencies. Verific's HDL parsers and elaborators can offer that opportunity, saving time and resources. RUSHC's Veriapps package provides a tool kit to complement the Verific HDL parsers for assembling EDA applications.

###

About RUSHC
RUSHC comprises a team of EDA engineers with extensive experience developing Verific-based applications.

About Andy Fox
Andy Fox is the principal of RUSHC (The Really Useful Software and Hardware Company) with offices in the United States and Armenia. His expertise ranges from complex FPGA and ASIC logic design and EDA tool development of logic synthesis, logic verification and algorithms for power optimization and physical logic optimization. He holds a Bachelor of Science degree in Electronics and Computing and a Master of Engineering degree in Microelectronics from the University of Durham in the United Kingdom, and a Ph.D. degree in Engineering from the University of Aberdeen in Scotland.

References:

[1] Verific: http://www.verific.com

[2] S. Sutherland and D Mills, "*Verilog and System Verilog Gotchas, 101 Common Coding Errors and How to Avoid Them.*"

[3] J. Cong and Z Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC formulation." Design Automation Conference 2006.

[4] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization, Master's Thesis," Computer Science Department, University of Illinois at Urbana-Champaign, Dec. 2002

[5] Freescale Semiconductor, Verilog HDL Coding Semiconductor Reuse Standard.

[6] Qi Zhu, N. Kitchen, A Kuehlmann, A. Sangiovanni-Vincentelli, "*SAT Sweeping with local observability don't cares.*"

[7] A Mischenko, *"ABC, A System for sequential synthesis and verification." http://www.eecs.berkeley.edu/~alanmi/abc/*

[8] V Ganesh, D Dill. "*A Decision Procedure for Bit-Vectors and Arrays*." Proceedings of Computer Aided Verification 2007.

[9] L. De Moura, N. Bjorner. "*Z3: an efficient SMT solver.*" TACAS'08/ETAPS'08 Proceedings of the Theory and Practice of Software. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems