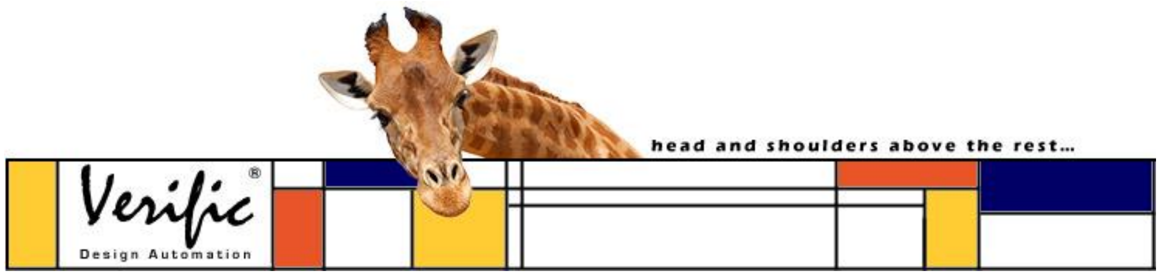


Exploiting Verific tools and features at the right abstraction level

Andy Fox, Edvard Ghazaryan, Hovhannes Ter-Milqsetyan, Tigran Sargsyan and Vigen Gasparyan
www.rushc.com

First published on Tech Design Forum, Aug 29, 2014

EDA vendors and internal CAD teams use Verific parsers for tool development. Here's how one company developed its strategy for this popular technology.

Introduction

Verific Design Automation specializes in Verilog, VHDL and SystemVerilog language processing sub-systems. Its users develop software where Verific-based technology serves as the front end for a wide range of EDA and FPGA design tools. These tools are used during analysis, simulation, verification, synthesis, emulation and test.

This article discusses the use of Verific technology by our team at the Really Useful Software and Hardware Company. We hope these experiences will be of use to the many other Verific users out there and we also describe our own efforts to extend the technology with a series of 'apps' addressing common tool developer issues.

Before getting into our use strategies, we need to start with a quick backgrounder on Verific's offering.

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

- Parse tree
- Statically elaborated parse tree
- Operator netlist
- Gate level netlist

Determining how best to choose between these levels will be one of the main themes of this article, but first we need to look a little closer at each type.

Parse tree-based elaboration

The parse tree representation is generated during analysis and traversed using a walker. A range of walker templates ensures that each syntax category can be easily traversed. Static elaboration further expands the parse tree by resolving parameters as well as generating statements and other statically determinable aspects of the HDL.

So, as an example, a developer can use the ‘VhdlExpression’ walker class to add a type checker that catches any mismatch in the sizes of a VHDL relational operator. The Verific code elaborates the walker so that the developer can get a type for any identifier reference. Each identifier instance is represented as an ‘IdReference’ that in turn refers to an ‘IdDefinition’ field from which all type information about the identifier can be extracted.

The parse tree can be structured to resolve data types as necessary for, say, such instances as the evaluation of recursive functions or the resolution of dynamically assigned ranges in VHDL. Mixed-language support is provided using a ‘vl_types.vhd’ package that permits various type conversions.

Operator netlist and gate-level netlist elaboration

Full elaboration translates a parse tree to a netlist by way of synthesis. With a little care, operators can be preserved and a bus-oriented or unflattened data model extracted. This is called the ‘operator netlist’.

The operator netlist is useful because it provides the semantics of the language via operators, including decoders, shifters and state elements. It also has a wide operators function for bus level structures (this function is known as a ‘netbus’ or ‘portbus’ in Verific terminology).

The same netlist view is generated for all languages. This provides a good starting point when a project includes tasks such as writing a simulation accelerator applicable to synthesizable designs written in any supported HDL.

Moreover, because Verific takes care of the language processing, the developer needs only to understand the netlist. A Verific netlist itself follows the traditional and familiar Electronic Design Interchange Format (EDIF)-type hierarchy of library, cells, instances, views and ports.

Using one of the APIs in the Verific toolkit, the user can flatten the operator-level netlist to primitive gates, such as PRIM_AND, PRIM_OR and PRIM_XOR. At this fourth level of elaboration, the full logic of the design is available.

The various abstraction levels are also summarized in Figure 1.

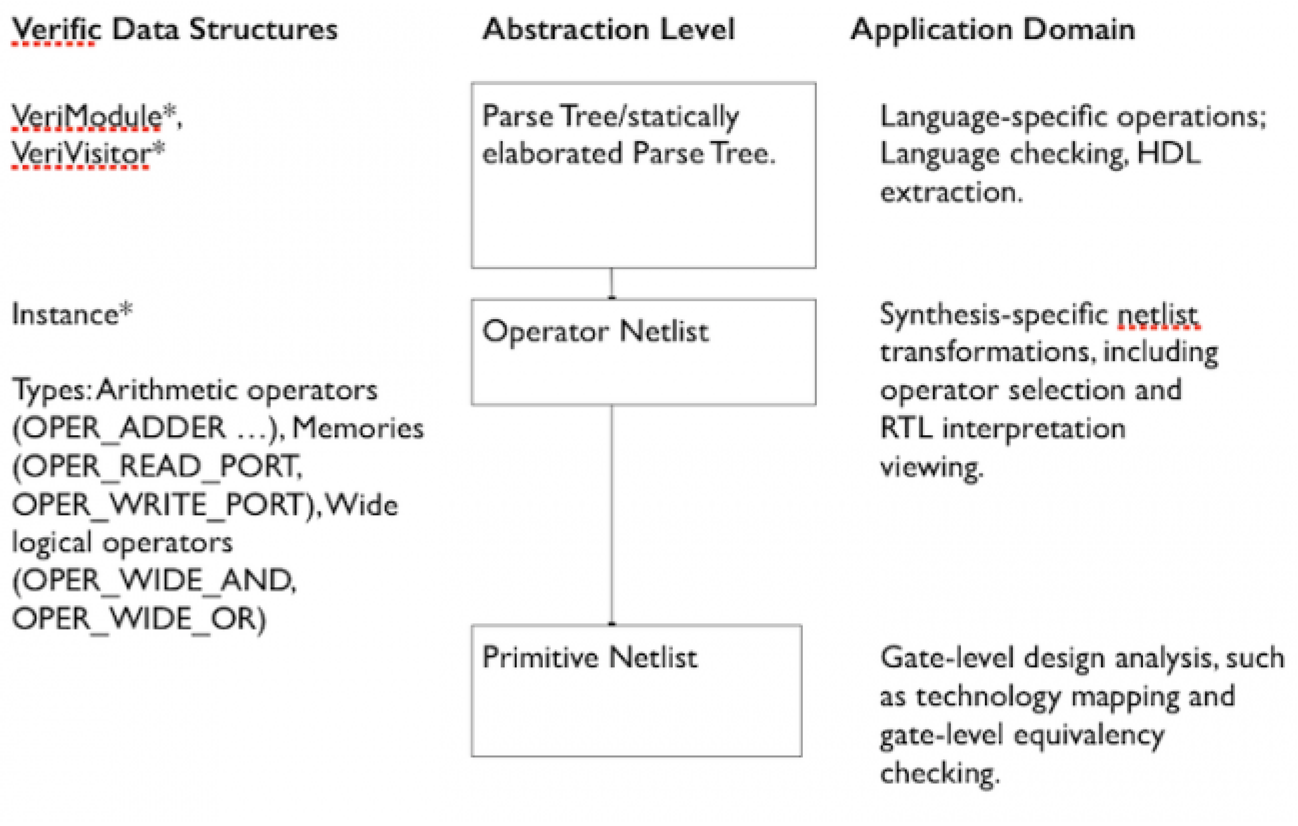

Verific Data Structures
Abstraction Level
Application Domain
VeriModule*, VeriVisitor*
Parse Tree/statically elaborated Parse Tree.
Language-specific operations; Language checking, HDL extraction.
Instance*
Types: Arithmetic operators (OPER_ADDER ...), Memories (OPER_READ_PORT, OPER_WRITE_PORT), Wide 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 Verific abstraction levels (Source: Verific/RUSHC)

Integrating Verific into a software flow

Once the appropriate Verific libraries have been linked, Verific's C++ APIs can be called directly from a developer's code. The APIs used most often are'Analyze' and 'Elaborate'. The Verific data structures are usually entered through either the top-level parse tree module (Verific API: veri_file::GetTopModules) or the netlist (Netlist::PresentDesign()).

At this point, most developers translate Verific structures to their own databases and continue independently of the Verific code base. However, the Verific netlist is stable enough to use in the developer's environment and that is our focus here. The challenge then lies in enabling such use so that the developer's in-house code base investment is protected while the Verific code is safety exploited also.

At RUSHC, we have run Verific evaluations based on three strategies that address this challenge:

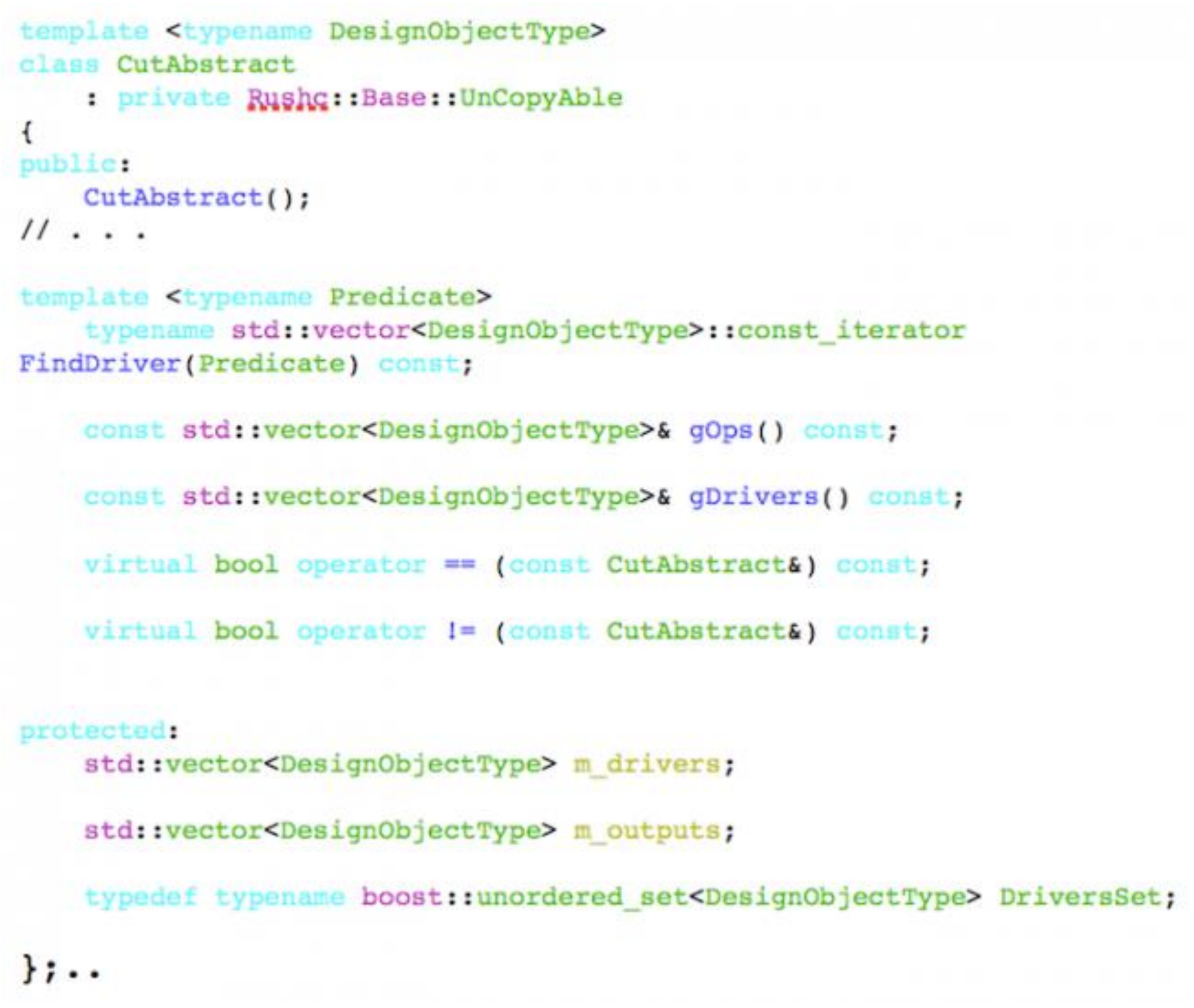
Deriving the Verific classes from a generic abstract class

Deriving customer-specific classes from the Verific classes

Template-based strategies. Creating algorithms parameterized by abstract types that provide concrete interfaces and use 'hardened' Verific types.

Options (a) and (b) allow all Verific APIs to be accessible natively from customer code. But it is option (c) that offers the broadest set of use cases by quickly allowing core algorithms to be ported to new structures. This option also keeps the style adopted by boost libraries.

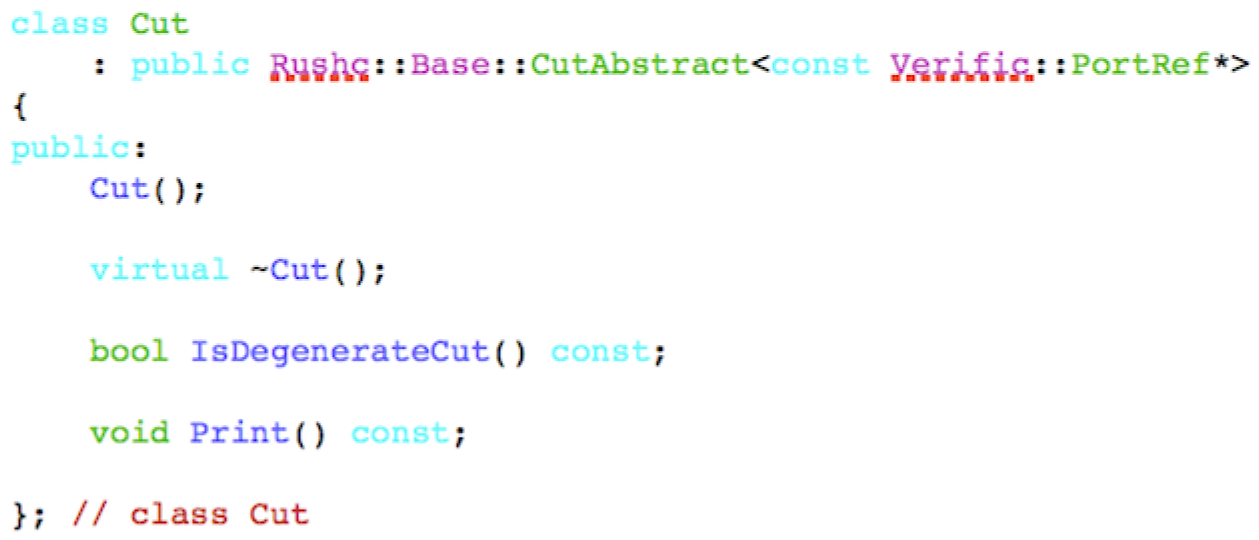
Consider this representation of a ‘cut’ in a netlist, the cut being nothing more than a group of pins delineating a region of a netlist. The abstract class for a cut is shown in Figure 2.



```
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;

};..
```

FIGURE 2 Abstract class for a cut (Source: RUSHC/Verific)

Note that the DesignObjectType is left as a type parameter. A concrete version of the cut and its various algorithms are then realized by hardening the type of the DesignObjectType to the Verific specific type ‘PortRef’, as shown in Figure 3.



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

    virtual ~Cut();

    bool IsDegenerateCut() const;

    void Print() const;

}; // class Cut
```

FIGURE 3 Cut and algorithms hardened (Source: RUSHC/Verific)

We used this approach to build a set of library routines and useful applications for formal verification and synthesis that make full use of the Verific database and that are broad enough for use on multiple structures. Specifically, RUSHC was able to port a full formal verification method approach for generating timing exceptions from the Verific database to a customer database in a matter of days.

Case studies emphasize abstraction choices

We found that the key to success lay in determining the best level of abstraction in the Verific flow: parse tree (statically elaborated or not), operator netlist or primitive netlist. Problems will arise if this decision is not thought through carefully. The following scenarios show how RUSHC made its choices on three different projects and could help you do the same.

1: A mapping solution

This 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 primitive netlist level of abstraction was chosen. This decision freed EDA developers from having to code HDL processing/elaboration. They could instead focus on innovation through algorithm development. The Verific netlist produced by elaboration was further improved by the SAT Sweeping technique for simplifying an AND/INVERTER graph (AIG) [6]. A template for a mapping and matching solution was devised that could be run on both the Verific netlist and the data structures used after place-and-route.

This project illustrated the value of creating templates and the stability and efficacy of the Verific database.

2: RTL acceleration for a synthesizable HDL subset

This project required the extraction of novel instructions (including MULT, LD, ADD, and STR) with hardware specific ones (e.g., TAP x [10:0]) to fish bits out of busses from an HDL for execution on a simulation hardware accelerator.

The client initially wanted to roll its own elaborator and was investing heavily to resolve complicated and generic HDL issues.

However, we determined that the project only needed the synthesizable HDL subset.

An operator level netlist elaboration was used as the basis for the machine instruction generation. Within a few weeks, the instructions were being generated.

The focus of the project shifted to establishing the value of high-speed simulation acceleration from resolving complicated HDL issues. Lesson learned: avoid becoming an HDL processing guru unless that is your core business.

3: Formal methods for timing exception checks

This project required the development and application of timing exception proving algorithms using formal methods such as AIG techniques.

From the start, data structures beyond the Verific netlist were required. So were Verilog counter examples and, where possible, references to the users' source HDL had to be inserted.

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 in 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 were devised for simulation, bounded model checking, and fixed-point analysis. As before, a template-based approach to algorithm development was used. Trade-offs were 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. Nevertheless, by adopting a template-based approach to development, the core algorithms were easily applied to both the company's database and the Verific database.

Going further

Our experiences at RUSHC led us to develop a wishlist of generic utilities for the Verific code base that it would be useful to develop. It included:

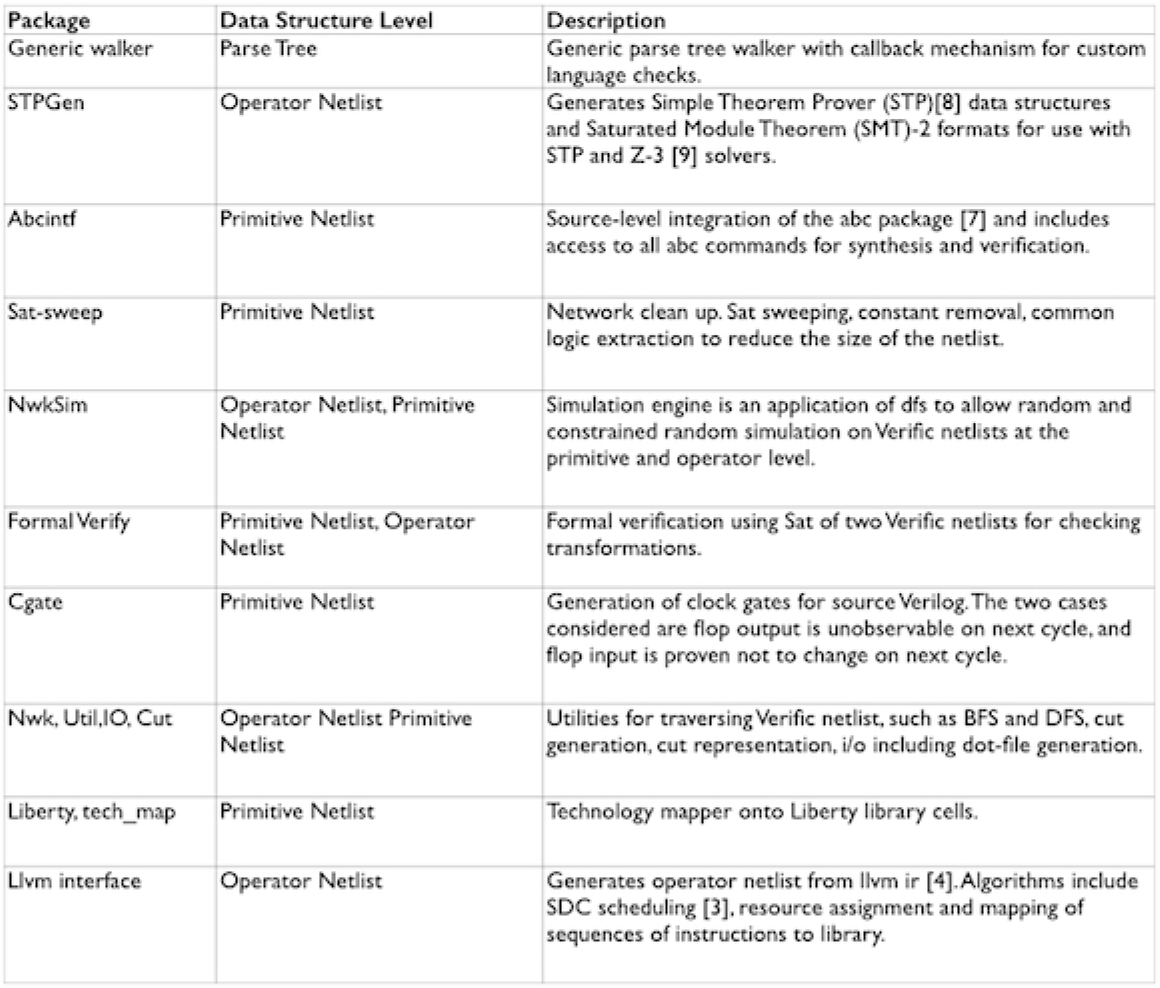
- A package for checking the parse tree for well known language gotchas and warnings. Every design team wants better error reporting.
- Redundancy removal and SAT-sweeping. Many applications need to start from the smallest netlist possible.
- Design Manager. The Verific APIs provide basic “analyze” and “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.

To fill some of these gaps, we have gone on to develop the Veriapps (for Verific Applications) package. This library of utilities operates on the Verific data structures. The packages shown in Figure 4 below are accessible via C++ or the Verific Perl interface, and the code is available in source form.

It is often said that you should stick to their core competencies. Verific’s HDL parsers and elaborators let those of us working on design software do just that, saving time and optimizing resources. Our hope now is that RUSHC’s Veriapps package will provides a tool kit that complements them, based on real-world experience with the technology.

About RUSHC

RUSHC comprises a team of EDA engineers with extensive experience developing Verific-based applications. It has offices in both the United Kingdom and Armenia. Find out more about the company at www.rushc.com.



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.

FIGURE 4 Currently available Veriapps (Source: RUSHC)