Progress Report on Development of VeriABC

Deliverable Report: SRC task 1875.001 - Aug 1, 2011
Task Title: Exploiting Synergy of Synthesis and Verification
Task Leaders: Robert K. Brayton and Alan Mishchenko
Students: Jiang Long and Baruch Sterin
Univ. of California/Berkeley

1. Overview

This report gives some details on our development of a front-end tool, VeriABC, for SystemVerilog/VHDL designs for both synthesis and verification applications. VeriABC interfaces with a commercial front-end parser and analyzer, Verific, to produce finite-state machine models. VeriABC processes the Verific generated netlist database to generate an AIGER model with box/bundle annotations representing high-level constructs and operations. The document describes the design and implementation of the tool and demonstrates the flow illustrated by some equivalence checking results.

As shown in Figure 1, we integrate VeriABC with ABC for formal verification. VeriABC processes the Verific Netlist Database to generate an AIGER model for downstream ABC verification engines.

2. Verific Parser Platform
Verific Design Automation [1] builds SystemVerilog and VHDL Parser Platforms which enable its customers to develop advanced EDA products quickly and at low cost. Verific's Parser Platforms are distributed as C++ source code or libraries and built on all 32 and 64 bit Unix, Linux, and Windows operating systems. Applications vary from formal verification to synthesis, simulation, emulation, virtual prototyping, in circuit debug, and design-for-test. We chose Verific as our front-end for its commercial success and stability in supporting the latest language constructs in SystemVerilog.

![Verific Parser Platform Architecture](image)

**Figure 2.** Verific Parser Platform Architecture

Figure 2 shows the architecture of the Verific parser front-end. The main commands we use in the Verific library are **analyze** and **elaborate**. **Analyze** creates parse-trees and performs type-inference to resolve the meaning of identifiers. The Parser/Analyzer supports the entire SystemVerilog IEEE 1800, VHDL IEEE 1076-1993, and Verilog IEEE 1364-1995/2001 languages, without any restrictions. The resulting parse tree comes with an extensive API.
Elaborate supports both static elaboration and RTL elaboration. Static elaboration elaborates the entire language, and specifically binds instances to modules, resolves library references, propagates defparams, unrolls generate statements, and checks all hierarchical names and function/task calls. The result after static elaboration is an elaborated parse tree, appropriate for simulation like applications. RTL elaboration is limited to the synthesizable subset of the language. In addition to the static elaboration tasks for this subset, it generates sequential networks through flipflop and latch detection, and Boolean extraction. The result after RTL elaboration is a netlist database, appropriate for applications such as logic synthesis and formal verification. This netlist database is the starting point of VeriABC and we utilize Verific provided C++ APIs to access the database.

2.1. Verific Netlist Database Structure

In this Section, we use Verilog terminology to present Verific's netlist database structures. The netlist database is rather intuitive and adheres to the original module and hierarchy definitions. Shown in Table 1, there is a one-to-one correspondence between the C++ API class definitions and Verilog constructs.

<table>
<thead>
<tr>
<th>Verific Database C++ API Class</th>
<th>Verilog RTL Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Netlist</strong></td>
<td>Module definition</td>
</tr>
<tr>
<td><strong>Instance</strong></td>
<td>Module instantiation</td>
</tr>
<tr>
<td><strong>Port</strong></td>
<td>Module port declaration</td>
</tr>
<tr>
<td><strong>Net</strong></td>
<td>Wire/reg/assign</td>
</tr>
<tr>
<td><strong>PortRef</strong></td>
<td>Port to Net connectivity</td>
</tr>
</tbody>
</table>

Table 1: Correspondences

A Netlist corresponds to module definitions in Verilog while an Instance object corresponds to module instantiation, after the module's parameters have been characterized. An Instance is a thin copy of the Netlist plus a pointer to its parent Netlist. A Netlist contains a set of Ports, Nets and Instances for its internal logic structure. A Port corresponds to the Verilog port definitions which can be input, output or inout. A Net is a named component, intuitively a wire. PortRef contains the connectivity between a Port and a Net. The direction of the PortRef can be in, out, or inout depending on the type of Port it contains.

Using these C++ objects, the Verific netlist database defines a directed hyper-graph and encapsulates the following types of information:

- **Design Hierarchy**: This is captured as an instance tree by the parent pointers in the Instance with a top-level netlist as the root.
- **Unique Hierarchical Name**: Following the design hierarchy through the instance tree, each internal object is assigned a unique hierarchical name.
- **Connectivity**: A directed hyper-graph is defined through Port, Net and PortRef: Port being the node, Net being the edge, and PortRef containing the connectivity and direction information between pairs of a Port and a Net. As an edge in the hyper-graph, a Net can be referenced in multiple PortRef objects.
• **Logic Definition**: At the leaf of the design hierarchy, a *Netlist* of primitive operator types such as *and*, *or*, *adder*, *flipflop*, *latches* etc defines the basic logic operators.

Recursively, the functional behavior of the design is captured through the directed hierarchical hyper-graph with basic logic operators at its leaf level.

3. **VeriABC: bit-level AIGER generation:**

VeriABC traverses the above netlist database and transforms it into a monolithic AIG which can be treated as a directed acyclic graph (DAG). The AIG contains primary input, primary output, register nodes and and-inverter nodes. Each named Port and Net in the Verific netlist has a mapped node in the AIG graph. Additional book-keeping information such as hash tables is created that map the hierarchical name to the corresponding AIG node. The down-stream model checker of ABC then reads in this AIG to conduct formal analysis.

3.1. **Architecture**

A hyper-graph represented in Verific netlist database structures is rather to hard traverse and conduct analysis/transformation at the same time.

![Verific Netlist Database](image)

**Figure 3.** VeriABC Architecture

As shown in Figure 3, we employ a two phase approach. First we construct an intermediate netlist graph, a DAG with extra annotated node types representing logic structure and connectivity of the flattened design. In addition to the simple node types in AIG graphs, extra node types contain annotations for language constructs such as tri-states, flipflops and latches etc. For example, flipflops contain set/reset pins and driving d-pins; latches contain additional gated-clock definitions. By language definition, a design can specify any signal for the clock and reset signals. Design behavior is defined by event-driven semantics. Further analysis needs be done to determine if there is an AIG representation that can capture the original design semantics. The intermediate netlist is a DAG for
which various traversal algorithms can be conducted for the later analysis. For this step, VeriABC only
traverses the design hierarchy and hyper-graph in the Verific netlist database to gather information and
construct the intermediate DAG representation without conducting any design style checking or
transformation.

3.2. Formal Modeling

The end result of VeriABC is an AIG model that is consistent with the original RTL design semantics.
AIG is characterized as a finite state machine model. Compared to the event-driven semantics in an
HDL language, the execution semantics of an AIG is synchronous with an implicit universal clock that
ticks at every step of the execution. The register loads in its driver value at the beginning of each step.
The semantics inferred from the Verific netlist structure is more complex, such as its flip-flops can
have arbitrary reset logic and clock networks. The task in formal modeling is to transform the above
design components into AIG registers with additional glue logic so that it maintains consistency. In its
simplest form, the following check determines if the design can be readily represented by an AIG.

- All registers are clocked by the same primary input signal which does not fanout to other nodes.
- Reset/Set signals are primary inputs
- No combinational loops
- No multiple drivers per node

More complex design modeling and transformations can be achieved by identifying certain patterns by
traversing the intermediate netlist graph. Our current implementation supports the above form and
produces a design style summary for debugging purposes. Although capacity and performance depends
on the type of individual transformation and analysis, for the above ones, the transformation and design
style checking is very fast, as it conducts only a few traversals of the intermediate netlist graph. After
design style checking, a final traversal of the intermediate netlist graph generates an AIGER file
representing the formal model.

3.3. Commands Implemented

VeriABC implementation utilizes the Tcl command interface shipped with the Verific library
distribution. The following is the list of commands implemented to manage the environment setup for
formal verification.

veriabc_analyze
   This command constructs the intermediate netlist graph in Figure 3 and conducts design style
   checking.

veriabc_set_reset
   Although a reset signal can be detected automatically in certain situations, this command
   provides the user with the option to specify the length of the reset sequence and phase of the
   reset condition. A user can also specify the initial value of the registers through a VCD
   waveform or textual file. In the generated formal model, the initial value of the registers will be
   valued at the end of the reset sequence and the reset condition will be disabled after reset.

veriabc_set_clock
A user can specify the clock periods and relationships in the situation when multi-clocks are in the design. A phase-counter network will be created in the formal model to generate the corresponding clock signals.

**veriabc_set_cutpoint**
This command will prune the cone-of-influence at cutpoint signal and treat each cutpoint as a free input.

**veriabc_set_constant**
This command sets either an input or a cut-point signal to a constant value.

**veriabc_set_assume**
This constrains a design signal to be a constant.

**veriabc_write**
This command writes out the final formal model in AIGER format. The above commands give a user the flexibility to model the environment with constraints and conduct design abstractions during verification.

### 3.4. Release distributions

A binary release of the VeriABC library (currently support 32 and 64 bit Linux) is available at [https://bitbuckets.org/lj/veriabc-alpha](https://bitbuckets.org/lj/veriabc-alpha). The release allows Verific users to build the VeriABC executable on-site with their own licensed Verific distribution. VeriABC development is done on Ubuntu Linux 10.04. Currently we have two active industrial users, one for verification and the other is mainly for synthesis. The backend engines in ABC are used independently.

### 4. Equivalence Checking Using VeriABC

One of our industrial collaborators uses VeriABC as the front-end for their in-house synthesis evaluation of various design prototypes. In their design flow, for each design, they produce various configurations with different target timing or area considerations. VeriABC and ABC are utilized to perform synthesis and mapping such that the results from different strategies can be compared in terms of area and delay. In one of these flows, they produce two versions of the same design, one is word-level and the other is flattened bit-level Verilog. Since both designs have the same number of registers and the inputs and outputs match respectively, a combinational equivalence setup is sufficient to check the equivalence of the two design versions (Note that the designs have simple clocks and reset logic, and all registers are initialized to zero by default). We use VeriABC to compile both versions of the design into AIGER format and uses the ‘cec’ and ‘dcec’ command in ABC to conduct equivalence checking.

However, for different configurations, the generated designs might differ in the number of registers and their input and output behaviors might not match exactly. In this situation, a sequential miter is created using ABC to compare the two AIGER model. After the miter is constructed, the bounded model checking command in ABC, ‘bmc3’, quickly reports the two designs are different by asserting the miter outputs. For more complex situations, ABC’s dsec and its suite of formal verification engines can be utilized to solve the sequential equivalence of the miter problem.
5. Annotating AIGER with Box and Bundle Information

During Verific compilation, a hierarchical design netlist is constructed and can be accessed through Verific Netlist Database. It contains information on design hierarchy, bit-vectors and certain word-level operators are preserved. This information is available before the flattening and AIGER construction. VeriABC captures this hierarchical information into boxes/bundles:

- **Bundle**: An ordered list of AIGER literals representing a bit-vector.
- **Box**: An list of bundles with input/output direction that marks the boundary of a box which corresponds to an instance in the design hierarchy.

The box/bundle is a bottom-up approach as it does not represent a design hierarchy directly. However, the full design hierarchy can be reconstructed by traversing these box/bundle definitions. The advantage of Box/Bundle annotation is that it is so simple that it does not require any fundamental change from the existing AIGER model. These box/bundle definitions are annotated later onto the AIGER graph. The following section describes how the Box/Bundle information is encoded on the existing AIGER model in the comment section.

The following Figure illustrates the connection between AIGER and box/bundle annotations. A multiplexer with 8-bit inputs/outputs in the AIGER graph is extracted using annotations. The annotation effectively groups a list of literals into a single bit-vector (bundle) object and box uses a set of bundles as input/output boundary definitions to retain the high-level design constructs. This annotation adds structural information onto the original “sea-of-gates” AIGER graph.

![Figure 4. ZAIGER Annotation](image)

The AIGER model is a popular bit-level format of an And-Inverter graph. Its basic objects are interpreted as:

- Primary inputs
- Sequential elements
- AND nodes
Each object has a unique id number corresponds to an unsigned integer. A literal is a signed form of the object id, representing if the corresponding signal is inverted or not. The AIGER model is a concise and simple bit-level representation used to represent sequential Boolean networks for use with verification and synthesis. AIGER is supported by ABC and many other academic and industrial tools; it is used as the input format for the HWMCC'10 (Hardware Model Checking Competition 2010).

The Zaiger format encodes the box/bundle annotation into the AIGER comment section. The overall extension of AIGER is proposed by Niklas Een and Alan Mishchenko. Zaiger adopts the same overall design methodology and append the encodings into the same AIGER file. The following C++ code demonstrates the information captured in ZAIGER:

```cpp
namespace ziger_ns{
    typedef aiger_ns::Lit Lit;
    typedef uint32_t ziger_bundle_t;
    typedef uint32_t ziger_name_t;

    static const ziger_bundle_t bundle.Undef = -1; // bundle with no definition
    static const ziger_name_t mid.Undef = -1; // name is not defined

    typedef enum ziger_dir_e {
        dir_IN=0, dir_OUT=1, dir.Undef=2
    } ziger_dir_e;

    class ziger_bundle_s: public vector<Lit> {
    // a bundle is a list of literals
    
    class ziger_bundle_t{
        ziger_name_t _nid; // index into ziger_s::names
        ziger_bundle_t _bid; // index into ziger_s::bundles
        ziger_dir_e _dir; // dir_IN or dir_OUT
    }

    class ziger_box_s : public vector<ziger_bundle_t> { // a list of directed bundles.
    char * _oper; // the operator definition ADD/MUL/MUX etc or UserDefined
    
    class ziger_s {
        vector<char *> _names;
        vector<ziger_box_s> _boxes;
        vector<ziger_bundle_s> _bundles;
    }
    
    
}
```

Please refer to the zaiger distribution for the actual encoding implementation of ZAIGER at https://bitbuckets.org/lj/zaiger

6. Control/Data Path Extraction and Analysis

One of the practical goals in our exploration of Verific netlist database in terms of box/bundles is to conduct data-path and control-path analysis. Intuitively this high-level abstraction information should gives insight for downstream tool flows in verification and synthesis. As a first step in the process, we would like to identify the boundary and sub-graph that isolate the corresponding data flow and control centric logic structures. The afore-mentioned box/bundle information captured in ZIGER format or in the original Verific netlist database is a hyper-graph, which is not convenient for program traversals for
analysis. Therefore we flatten the design and convert the Verific structure into a directed graph consisting of nodes for bit and bundle operators. This hybrid directed graph encapsulates bit-level and bundle-level information in one place and further analysis can be conducted on it. Book-keeping information based on unique hierarchical names is preserved such that there is a mapping from the bundles into the bit-level AIGER model generated as described in Section 3.

Currently we are in the process of developing this framework. The bit/bundle graph has been created and statistics have been collected for illustrations. In the DDR2 design from the OpenSparcT1 design, Table 2 lists the number of internal operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Count</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverters</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>1 bit AND</td>
<td>626</td>
<td></td>
</tr>
<tr>
<td>1 bit OR</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>1 bit MUX</td>
<td>806</td>
<td></td>
</tr>
<tr>
<td>1 bit Flipflop</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>ADDER</td>
<td>80</td>
<td>From 3-bit to 33-bit wide</td>
</tr>
<tr>
<td>REDUCE.OR</td>
<td>16</td>
<td>From 4-bit to 9-bit wide</td>
</tr>
<tr>
<td>REDUCE.XOR</td>
<td>4</td>
<td>Two 32-bit and Two 33-bit wide</td>
</tr>
<tr>
<td>Less than</td>
<td>9</td>
<td>From 3-bit to 32-bit wide</td>
</tr>
<tr>
<td>N to 1 MUX</td>
<td>4</td>
<td>All 5-bit wide</td>
</tr>
<tr>
<td>Bit-wise AND</td>
<td>141</td>
<td>From 2-bit to 35-bit wide</td>
</tr>
<tr>
<td>Bit-wise OR</td>
<td>55</td>
<td>From 3-bit to 35-bit wide</td>
</tr>
<tr>
<td>Bit-wise Inverter</td>
<td>1</td>
<td>4-bit wide</td>
</tr>
</tbody>
</table>

**Table 2. DDR2 design statistics**

Other operators such as MUX, EQUAL, MINUS etc, which operate on bundles of different width are not listed here.

7. Conclusions and Future Directions

With the development of VeriABC, we now have a tool that can be used to get access to realistic designs from both the public domain as well as cooperating companies. In addition, we can still preserve some high level information about the design hierarchy and the bit-vectors and operators. Our premise has been that this information can be quite useful in both synthesis and verification when the design has been flattened to the bit level (bit-blasted). Having this hybrid information allows us to combine the powerful verification and synthesis bit-level engines in ABC with the ability to selectively and iteratively abstract away large portions of the design that have little interaction with the control part. This is in contrast with other methods which do the high level transformations and then throw the design over the wall to the bit engine.

Now that we have the hybrid graph representing the control flow and data flow of the design in terms of boxes/bundles, it opens opportunities for a new round of research experiments utilizing the high-level information to help down-stream ABC in synthesis optimization and verification. This is the direction that our future research will continue along.