Toward GHDL_EVAL : A Framework for Deeply Embedding Simple HDLs in HOL

Michael D. Jones, Trent N. Larson and Phillip J. Windley*

{jones, larson, windley}@cs.byu.edu
Computer Science Department, Brigham Young University, Provo UT 84602, USA

Abstract. Netlist-based hardware description languages (HDLs) are common tools in engineering design. It would be useful if there were a general approach to express HDLs in a formal system, and tools to verify the behavior of components. In this paper, we describe a general method for embedding hierarchical netlist-based HDLs in HOL. We also describe a simulation engine, called GHDL_EVAL , which expresses the operational semantics of devices in such embedded HDLs. We then give an example of using GHDL_EVAL to verify a simple hardware device.

1 Introduction

We are working on the infrastructure for a distributed library of verified hardware components in HOL (for more detail see [?]). Because our target users are working engineers rather than formal methods experts, we seek to make use of existing tools and methodologies as much as possible. One way we do this is by supporting implementations described using existing hardware description languages (HDLs). Using HDLs inside the HOL proof assistant is not a particularly new idea [?] [?]. In our representation strategy, the HDL expressions are parsed into a HOL datatype which represents the abstract syntax of the language. Parse trees are interpreted by an HOL function definition using a library of primitive definitions. Such an embedding is usually referred to as a "deep embedding."

The novel feature of our particular embedding scheme is its extensibility. Throughout the design and implementation, we have maintained a level of abstraction and a degree of modularity that facilitate the use of various HDLs and primitive representations. We selected an abstract syntax that can be used to encode not just a variety of concrete syntaxes, but a variety of modular, netlist level HDLs. In addition, we have parameterized our simulator definition by the representation and evaluation of primitive devices. Thus allowing users to change the primitive representation by defining a new library–without modifying the simulator or abstract syntax. While the work described in this paper has been completed, there is much to be done to improve the efficiency and simplicity of our HDL support for libraries of hardware device development tools.

* This work was sponsored by the National Science Foundation under NSF grant MIP–9412581

The next section we describes related work. Section three describes interesting features of our embedding scheme in more detail. In section four, we present an example of a parser and primitive library used in GHDL_EVAL for simple verification. In section five, we discuss observations made during the project thus far. The final sections contains results and future work.

2 Related Work

Hand-made embedding of particular languages in the HOL proof assistant are becoming passe as tools which generate embeddings from higher-level descriptions (much like lex and yacc) become available and easy to use. Reetz and Kropf's formalized compiler generator [?] has been previously been used to deeply embed a significant subset of VHDL [?]. We do not use Reetz's generator because we intend to support a narrower class of languages thus allowing us achieve greater simplicity and efficiency by making more assumptions about our source languages.

Our abstract syntax is inspired by the formally defined DUAL-EVAL language developed at Computational Logic Incorporated (CLInc.) [?]. At CLInc, DUAL-EVAL was developed for use in the Boyer-Moore theorem prover and was used to describe the FM9001 microprocessor netlist. The FM9001 description was used to verify the netlist description against a high-level specification and to fabricate the working chip. We selected DUAL-EVAL as the source for our abstract syntax because it was designed for use with a library of primitive devices rather than a fixed set of native primitives.

The GHDL_EVAL simulator defined in HOL makes extensive use of the definition libraries developed by Melham and Gunter. Melham's library supports primitive, simple recursive type and function definitions [?]. Gunter's library allows mutually recursive primitive type and function definitions [?]. Both libraries provide excellent tools for developing our simulator and hide the complexity of proving existence and well-foundedness properties.

3 Implementation

We have implemented GHDL_EVAL in HOL90.8 using the contributed mutual recursion package and the recursive definition library. The implementation can be divided into parts supplied by the GHDL_EVAL package and parts supplied by the user. We discuss tools provided by GHDL_EVAL first, then present a description of user-supplied tools.

3.1 Tools Provided by GHDL_EVAL

GHDL_EVAL consists of two major parts: an abstract syntax for hierarchical netlists, and a function for simulating the outputs and state changes of a netlist device.

An Abstract Syntax for Netlists A HOL datatype is used to represent the structure of HDL expressions within HOL. The datatype is mutually recursive and contains constructors for modules, primitives, and occurrences. Modules represent devices built from occurrences of primitives or other modules. Primitives contain the behavior of the least abstract hardware device in a given model. Although there is a primitive constructor in the abstract syntax datatype, the actual representation and interpretation of primitive devices is left up to the user.

The type-signatures for devices (either a primitive or a module) and occurrences are as follows:

```
primitive: <name> -> <input names> -> <output names> -> <state names>
           <output functions> -> <state functions>

module:    <name> -> <input names> -> <output names> -> <state names>
           <occurence list>

occurrence: <name> -> <input names> -> <output names> ->
            <device>
```

A Device Simulation Engine The simulator definition takes the netlist, a device name, and the initial states and input values as parameters. Definitional rewrites are used to "run" the simulator, and each "run" corresponds to a single tick of time. Therefore, to model the behavior of a stream of inputs on a sequential circuit, the user would provide the initial states and run the simulator once for each time tick. Each run would require the state values from the previous run as well as the inputs for that time tick as parameters.

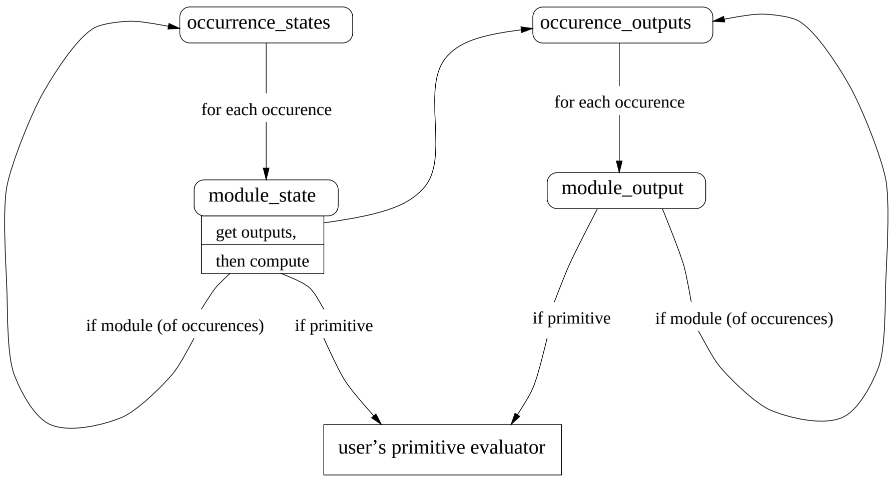

Fig. 1. How GHDL_EVAL 's simulator works

Figure ?? illustrates the recursion in the `GHDL_EVAL` simulator. The labeled arrows represent the flow of control, handled via pattern matching. Rounded boxes contain subfunctions of the simulator, and square boxes contain further descriptions of those. For example, `occurance_states` computes the state of an occurance by calculating the `module_state` for each module in the occurance. Furthermore, a `module_state` is computed using either the supplied primitive simulator, if the module is a primitive device, or the occurance_states simulator if it not primitive.

3.2 Tools Provided by the User

`GHDL_EVAL` makes no assumptions about the representation or interpretation of primitive values nor the concrete syntax of the source HDL, thus allowing the user great flexibility. As a result, the user must supply a parser which outputs `GHDL_EVAL` abstract syntax and a primitive library which contains primitive devices and their interpretation. Concrete examples of user-supplied parsers and primitive libraries, along with their use, are given in the next section.

While developing new parsers is relatively straightforward, creating new primitive libraries is more difficult. It is anticipated that a set of primitive libraries will be developed and made available to allow verification using a wide variety of devices.

3.3 Parser

A user supplied parser generates abstract syntax from some source HDL. The parser may be implemented in any language using any tool at hand. We include various SML-to-HOL functions so the parser may generate SML output and `GHDL_EVAL` will automatically generate the corresponding netlist in HOL.

3.4 Primitive Library

The primitive library contains primitive device definitions and functions used to lookup and evaluate the definitions. The primitive evaluator definition should accept a primitive device and a list of primitive input values and states, and return a list of output values and states. An important aspect of a primitive device is the representation of behavior. Each primitive should contain some expressions which, when evaluated, correctly and efficiently model the behavior the device. The interface between the primitive evaluator and `GHDL_EVAL` is straightforward. It requires three type definitions, one function definition, and one tactic:

1. *The type* `inputtype`, defining the collection of inputs to any device.
2. *The type* `outputtype`, defining the collection of outputs from any device.
3. *The type* `functiontype`, defining the primitive function objects that determine the outputs from the inputs.

4. *The function* `eval`, the HOL function that takes two arguments (of type functiontype and inputtype) and evaluates to a result (of type outputtype).
5. *The tactic* `EVAL_TAC`, which resolves `eval` calls into results of type outputtype.

4 Example

This section contains an example of both a parser and a primitive library along with their use in `GHDL_EVAL` . The parser was generated for the BOLT hardware description language. The primitive library models AND, OR, and XOR gates operating on a three-valued logic (containing true, false and unknown). The parser and primitive library are used with `GHDL_EVAL` to verify the behavior of a half-adder implemented at the gate-level in BOLT.

4.1 Parser

The parser in this example was written for the BOLT HDL. BOLT is a hierarchical HDL developed at AMI and supported by a variety of existing tools. A brief overview of BOLT syntax follows.

Devices in BOLT are described by `MODULE`s. Modules have three parts:

- *The module header.* This contains the module's name and all the input and output names:

```
MODULE  <output name list> . <module name> <input name list>;
```

- *Parameter declarations* (optional). The only available parameters are `OUTPUT_CODE` and `STATE_CODE`. (Using either of these parameters makes the device into a primitive device.) Parameters are declared with `PARAM`, followed by the parameter name and the value of the parameter.
  Primitive modules are created by assigning the `OUTPUT_CODE` or `STATE_CODE` parameters to a primitive function representing the device's behavior. The behavior function appears in quotes: `PARAM OUTPUT_CODE = "<primitive function calls>"`. The structure of a primitive function call parallels the structure of the module header:

```
<output name list> . <primitive function> <input name list>;
```

- *The module body.* This contains occurrences of other modules surrounded by `BEGIN` and `END;`. (The module body is empty if this module is a primitive; see *Parameter declarations* above.) The structure of occurrences also parallels the structure of the module header:

```
<output name list> . <module name> <input name list>;
```

The BOLT parser is a top-down recursive descent parser built from a collection of SML functions, one for each non-terminal of the BOLT grammar. When given a file containing BOLT modules, the parser returns the parse tree of those modules in the `GHDL_EVAL` abstract syntax. This parse tree is passed as an HOL term to the simulator as the netlist of devices.

4.2 Primitive Library

Before choosing a primitive function representation for the first `GHDL_EVAL` library, we investigated different evaluation strategies for the primitives. We first experimented with a very general evaluator by embedding a subset of SCHEME in HOL using environments to track variable values. However, we concluded that such a strategy is too slow and complicated. On the other hand, storing functions as lookup tables was fast and simple but required too much space (especialy for a 32 input multiplexor). We settled on a tree based representation very similar to binary decision diagrams. In this representation, we use the function inputs to guide a traversal of the tree until reaching an output at a leaf node. A tree representing the boolean OR function on three valued logic is given in figure ??.

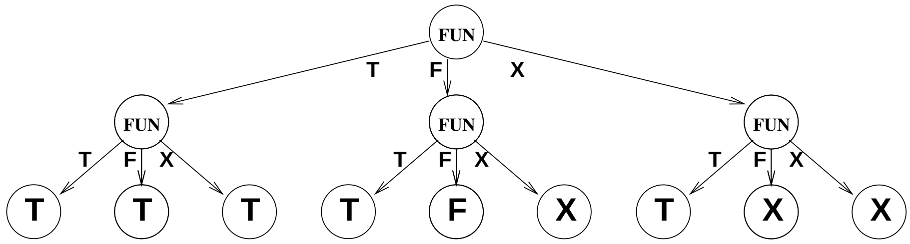

Fig. 2. Representation of an OR function in the primitive library.

In the current version of `GHDL_EVAL` , we represent the gate-level behavior of devices with the familiar `T`, `F`, and `X` of three-state logic as predefined primitive constants. The primitive functions on these constants are `and`, `or`, and `not`.

The underlying representation is only important to the parser and the `eval` function that the primitive library provides. For example, a primitive definition for the formula: $out = in1 \wedge in3$ appears as: ``out . and in1 in3;'' in the `OUTPUT_CODE` or `STATE_CODE` string in BOLT. (See section ??.) The parser translates this simple HDL formula into the `functiontype` arguments for use inside the simulator.

BOLT Half-Adder Definition In this section, we illustrate the BOLT syntax and verification using `GHDL_EVAL` with a common half-adder. To verify a half-adder implemented in BOLT using `GHDL_EVAL` , the file containing the BOLT description is parsed and the netlist abstract syntax is bound to a HOL constant definition.

In the BOLT file, the `and2` and `xor2` gates are defined as primitives and the `half-adder` is defined as a module as shown below.

```
MODULE out .and2 input1 input2;
   PARAM
 OUTPUT_CODE = "out .and input1 input2;"
   BEGIN
   END;

MODULE out .xor2 in1 in2;
   PARAM
         OUTPUT_CODE = "out .(or (and in1 (not in2))
                                 (and (not in1) in2));"
   BEGIN
   END;

MODULE carry sum .half_adder a b;
   BEGIN
         sum .xor2 a b;
         carry .and2 a b;
   END;
```

The following call to `name_netlist` binds the netlist abstract syntax (generated by `load_hdl`) to the name "nlist" in a HOL term.

```
name_netlist "nlist" (load_hdl "filename.bolt");
```

BOLT Half-Adder Verification Before proceeding with the complete verification of the half-adder on arbitrary inputs, we present a half-adder simulation on specific inputs to demonstrate the simulator operation. The following goal will simulate the behavior of a half-adder on the input values `[T, T]`.

```
simulate avalue "half-adder" [("a",T);("b",T)] [] nlist
                                     = [("carry",T);("sum",F)]
```

In this goal, we verify that on the inputs `[(''a'',T);(''b'',T)]` the output of the half-adder is equal to `[(''carry'',T);(''sum'',F)]`. Rewriting with the definition of `simulate` results in the application of `simulate_device_output` to the proper parameters. Further rewrites and conversions reduce function applications and expand let expressions until the goal is proven. All of the rewrites and conversions needed to prove simple goals such as this one have been bundled into `SIMULATE_TAC`.

To verify the half-adder on all inputs and outputs, the call to the `GHDL_EVAL` simulator is postulated as a goal.

```
! in1 in2 carry out.
      (valid in1 / valid in2) ==>
       ((simulate avalue "half_adder" [in1; in2] [] nlist =
                                                  [carry; out])
        =
        (half_adder_spec in1 in2 sum carry))
```

Since we are using a three-state logic with an unknown state, `valid` ensures that the input values are well-defined. The goal is proven using a series of rewrites and conversions and a case split on the input variables. The order of the rewrites and conversions and the timing of the case split are have a large impact on the amount of time and space required to prove the goal. We have provided several tactics to solve this problem.

The first step in the proof is to simplify the expression as much as possible for performing the case split. The tactic `PRE_SIMULATE_TAC` performs this simplification. After simplifying the goal, a case split is done on the input values. A predicate is used to filter out cases involving inputs with the unknown value leaving us with four subgoals.

Next, we reduce primitive function applications and rewrite with the specification to further simplify each of the subgoals. At this point in the proof, each subgoal has the form

```
(--`(F = out) / (F = carry) = (F = carry) / (F = out)`--)
```

and can be easily solved. Proving each of these subgoals completes the verification.

5 Results

We have performed a deep embedding of an abstract syntax for netlist level heirarchical HDLs in HOL90 which is parameterized by the primitive devices used to represent the meaning of HDL constructs. We have also developed a parser for the BOLT HDL to the `GHDL_EVAL` abstract syntax and a primitive library containing simple gates using three-valued logic. The `GHDL_EVAL` tool along with the parser and primitive library have been used to verify simple combinational and state-holding circuits.

In performing the embedding, we had two goals, both of which we believe (but have not yet demonstrated) we have accomplished:

- To create and embedding flexible enough to accomodate various simple heirarchical HDLs. We believe the abstract syntax we've defined in HOL will accomodate the family of languages we seek to support. To date, we have only tested it with the BOLT language.
- To create an interpreter parameterized by primitive devices. Parameterizing the interpreter on the primitive representation and interpretation also appears to be successful although we have not yet satisfactorily tested it. Thus far, we have used primitives operating on two and three valued logic and used

a variety of primitive function representation and evaluation schemes. These reletively minor changes were quite easy (when performed by the people who created the system).

Due to its utter simplicity, GHDL_EVAL does have its limits. The abstract syntax is not expressive enough to encode languages like VHDL and Verilog. Also, the current simulator is quite inefficient and verifications much more complex than a full adder modeled at the gate level overwhelm the simulator.

6 Future Work

To demonstrate that GHDL_EVAL is suitable for use with various HDLs and primitive libraries, we will develop larger examples of parsers, libraries and verifications. Our next source language will be the netlist components of HML. HML is a polymorphic, strongly typed HDL based on SML [?].

We are also working on a primitive library using the VHDL Standard Logic Package with 46 unique signal values. This evaluation scheme in this library will be more complex and use the signal resolution function designed by Gambles and Windley [?].

In addition to further testing GHDL_EVAL we intend to make GHDL_EVAL more efficient and easier to modify. It is anticipated that we will:

- Develop a tool for creating new libraries.
- Restructure the simulator to improve efficiency.
- Perform more complex verifications to identify weaknesses in the system.
- Create tactics to handle more general proofs about devices and combinations of devices.

By embedding the simulator in HOL, we can not only evaluate and validate particular implementations of devices; we can also prove theorems about certain combinations of hardware devices. Hardware combinators are algorithms which describe how to build a device from more basic components [?] [?]. (These algorithms usually describe a repetition of the components, such as a word-size adder constructed from 32 full-adders.) Once certain properties of the basic components are proven, we hope to can prove useful facts about such combinations of devices.

It may turn out that simulation is not the right approach to representing the semantics of hardware devices in HOL. Preliminary examples indicate that the state explosion encountered in simulation based verification will become prohibative for even simple devices. Future work will perhaps be directed at representing rather than simulating devies in the HOL logic.

Finally, our primary interest in creating GHDL_EVAL was not GHDL_EVAL itself, but rather GHDL_EVAL 's use as the input mechanism and proof support for a library of abstract, verified hardware components. Once GHDL_EVAL is sufficiently developed, we will turn our efforts to the problem of representing and retrieving such components (and their proofs), which will be organized by type and stored in a library distributed across the internet.

References

1. Warren Hunt Jr. Bishop Brock and William Young. Introduction to a formally defined hardware description language. In *Theorem Provers in Circuit Design*, 1992.
2. Mark Longley F. Keith Hanna, Neil Daeche. Specification and verification using dependant types. *IEEE Transactions on Software Engineering*, SE-16(9), September 1990.
3. Jody Gambles and Phillip J. Windley. Integrating formal verification with cad tool environments. In *Proceedings of the Fourth Annual IEEE/NASA Symposium on VLSI Design*, October 1992.
4. Jody Gambles and Phillip J. Windley. Reasoning about the vhdl standard logic package signal data type. Technical report, Laboratory for Applied Logic, 1994.
5. Elsa J. Gunter. A broader class of trees for recursive type definitions for hol. In Jeffery Joyce and Carl Seger, editors, *HUG 93: HOL User's Group Workshop*, 1993.
6. Miriam Leeser Mark Aagard John O'Leary, Mark Linderman. HML: A hardware description language based on SML. In *Proceedings of the IFIP Conference on Hardware Description Languages (CHDL 93)*, Ottowa Canada, 1993.
7. T. F. Melham. A package for inductive relation definitions in hol. In Karl N. Levitt Myla Archer, Jeffrey J. Joyce and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Aplications.* IEEE Computer Society Press, 1991.
8. Thomas Kropf Ralf Reetz. Simplifying deep embedding: A formalised code generator. In Juanito Camilleri Thomas F. Melham, editor, *Higher Order Logic Theorem Proving and Its Applications*, number 859 in Lecture Notes in Computer Science. Springer, September 1995.
9. Ralf Reetz. Deep embedding vhdl. In James Alves-Foss E. Thomas Schubert, Phillip J. Windley, editor, *Higher Order Logic Theorem Proving and Its Applications*, number 971 in Lecture Notes in Computer Science. Aspen Grove, Utah, Springer, August 1995.
10. Mike Gordon John Harrison-John Herbert Richard Boulton, Andrew Gordon and John Van Tassel. Experience with embedding hardware description langauges in hol. Technical report, Hardware Verification Group, Cambridge University.
11. Phillip J. Windley. Correctness properties for iterated hardware structures. In *Proceedings of the Fifth Annual IEEE/NASA Symposium on VLSI Design*, November 1993.
12. Phillip J. Windley. A distributed, type-based library of abstract hardware modules, October 1994. Proposal to the National Science Foundation.