

# Extending The Design Process With Formal Verification Technology

Edward P. Stabler  
Professor, Computer  
Engineering  
Syracuse University  
(315)443-4370  
stabler@cat.syr.edu

Michael P. Nassif  
Rome Laboratory  
(315)330-3342  
nassifm@rl.af.mil

Robert J. Paragi  
Rome Laboratory  
(315)330-3547  
paragir@rl.af.mil

## **Abstract**

This paper describes an experiment demonstrating the value of computer-based reasoning in hardware design. An interactive theorem prover is used to prove that VHSIC Hardware Description Language (VHDL) designs meet their specifications. Hardware designs and a formal logic notation are related by way of a formal semantics for VHDL developed by Rome Laboratory under a contract to Odyssey Research Associate Inc. (ORA). The advantage of starting with a formal specification is that the specification is precise and compact. In hierarchical design the higher level designs depend only on the specifications of components, not their architecture. Each refinement towards an implementation may be traced back to the original specification for compliance, i.e. verification. In verification it is not necessary to reason about delta cycle activity. This paper describes the process of performing a formal proof that a VHDL design satisfies its specification. The experiments make use of the ORA Larch/VHDL proof environment.

## **Introduction**

In response to the Computer Aided Design (CAD) community's need to handle larger and more complex designs and the need to be able to electronically exchange design information, a "standard" Hardware

Description language (HDL) was developed. The VHSIC Hardware Description Language (VHDL), was the HDL developed and established as an IEEE standard for the design and documentation of digital electronic systems.

Today however, verifying a large circuit's correctness by simulation remains impractical. Determining the correctness of a design by exhaustive simulation is not a practicable methodology due to the cost and time required to generate the test vectors and simulate the model.

In contrast to simulation, formal verification of hardware is a mathematical proof that the design of a digital circuit satisfies certain properties regardless of the values of the inputs. Formal verification tools can be used to prove that hardware designs satisfy properties such as functional correctness, security properties, and timing correctness.

## **The Larch/VHDL Solution**

A contractual effort is currently under way between the US Air Force's Rome Laboratory and Odyssey Research Associates Inc. (ORA) to develop a formal hardware verification environment [2]. The goal of this effort is to develop the capability to logically prove that a VHDL hardware design has certain properties, i.e., functional

correctness, security or timing, over all possible input combinations. Likewise, it is essential that there exist an ability to prove the absence of certain properties, e.g. deadlock, or resource contention.

ORA is leveraging off an Ada verification environment they have developed, known as Penelope [4]. Penelope is based upon the Larch two-tiered specification language developed at the Massachusetts Institute of Technology (MIT). The first tier, the Larch Shared Language (LSL) [1], is a first order predicate calculus used to build the traits, or theories, that define the sorts (or in the case of VHDL, types) used by the target language, i.e., bit, word, string, arrays, integer, etc. The second tier, called the interface language [3], defines the communication mechanisms of the target language, Ada, C, C++ or in this case VHDL, in the Larch notation. LSL is used to mathematically model data objects and operations on those objects, while the interface language maps the VHDL model into the abstractions represented by the Larch expressions for the purpose of formal reasoning.

Larch/VHDL is an interactive environment that helps its user to develop and verify digital electronic hardware designs written in VHDL. Larch/VHDL is well suited to developing code in the goal-directed style advocated by Gries [3] and Dijkstra [1]. In this style the designer develops a VHDL model from a specification in a way that ensures the VHDL model will meet the specification.

Penelope supports a window interface environment with several advanced features for entering specifications, developing code, and providing access to the Penelope theorem prover. As Larch or VHDL code is entered it is parsed and semantically analyzed. Also, as the specifications or VHDL change the prover automatically analyzes and re-evaluates the related components such as code or proof steps. Hence, the designer is continuously updated to the effects of

design changes to the specification or VHDL code.

An outline of the entire process is illustrated below in figure 1.

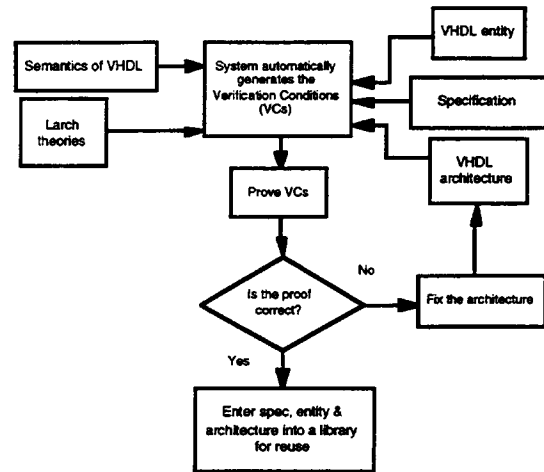


Figure 1. VHDL Formal Verification Process

The Larch/VHDL verification process augments the design process in four ways:

1. By developing Larch specifications which are independent of technology and implementation details:
  - A. Specifications are unambiguous, concise and immune to errors in translation from one natural language to another.
  - B. Specifications may be proven to be correct.
  - C. Specifications may be combined to form new specifications.
  - D. Specifications may be implemented in hardware or software.
2. By verifying the correctness of a VHDL model.

Determining the correctness of a design by exhaustive

simulation is not a feasible methodology due to the cost and time required to generate the test vector set and simulate the model. Formal verification of a hardware design can increase the designer's confidence that a digital circuit satisfies certain properties by reasoning over every possible input condition.

3. By verifying multiple implementations of a design in VHDL.

In many cases, several VHDL architectures may be developed in order to perform trade-off analysis. While multiple models will conform to the same entity interface declaration, their hardware implementations will vary in cost, area or performance.

4. By supporting hierarchical verification.

The Larch/VHDL methodology supports a form of verification of single components or cells of a design which is done earlier in the development cycle than simulation is typically done. Once verified, these components are available for reuse in other designs.

### Larch/VHDL

The Larch/VHDL environment includes a large body of traits that define the basic constructs of digital design such as bit, vector, gate, logic operations and so on. Traits define sorts (logical types) and state properties or assertions that must hold true. Traits also contain theorems which are statements that are deducible from assertions, previously deduced theorems, and/or the assertions or theorems of other traits that are included. The two-tiered Larch approach allows designers the capability to extend the library of traits in order to support user defined sorts in their models. Once implemented, the traits are available as library

components for reuse in other applications.

Traits are used to capture the concepts and relationships used in digital design. There are traits devoted to arithmetic concepts, and to data structures such as arrays and lists. To support VHDL semantics there are traits defining signals, and signal delay, and other concepts needed to express the semantics of VHDL. There are traits that describe the relationship between bit level operations and their arithmetic interpretation, in twos-complement or unsigned bit-level representations.

For example in the trait, **Bit2IntDefs** (Bit to Integer Definition) the interpretation of a Bit value as an integer is given as:

```
--| library lib;
--! Verification status: Verified
--| Larch
  Bit2IntDefs: trait
    Bit enumeration of '0', '1'
    introduces
      int: Bit -> Int
    asserts
      forall b:Bit
        bit2int: int(b) = (if b = '1'
          then 1 else 0)
--| end Larch
```

The trait **Bit2IntDefs** is a primitive trait because it does not include another trait and declares by enumeration sort **Bit** to have only two possible values '0' or '1'. The **introduces** declares the signature of the function **int** to have an input of sort **Bit** and return a value of sort integer. Signatures in a specification trait are analogous to function prototypes found in the declaration section of higher-level language (e.g. VHDL) programs, where a function name and its parameter list of inputs and output are given. There is one assertion, following the key word **asserts**, in this example:

```
forall b:Bit bit2int: int(b) = (if b =
'1' then 1 else 0).
```

A second trait, **Bit2Int**, uses the above trait and provides some properties, as

theorems, of the integer interpretation of bit values.

```
--| Larch
Bit2Int: trait
  includes (Bit2IntDefs)
  implies
    forall b:Bit
      bit2int_range: 0<=int(b) and
int(b)<=1
      bit_0_or_1: int(b) = (if int(b)
        = 1 then 1 else 0)
```

It is the purpose of the Penelope theorem prover to assist the user in proving theorems from the supplied axioms and included traits.

The following excerpt from a trait states the defines 'xor' on bit values using previously definitions of 'not', 'and', and 'or'. It is just a boolean algebra identity.

```
bit_xor: ((b xor c)=(((not b) and c)
or (b and (not c))))
```

Many of the theorems available to the designer are aimed at simplifying the proof process by substitution of simpler equivalent forms. The theorem shown below helps to simply 'xor' expressions when the values of the bit signals b, or c is known.

```
xor_conv (rewrite) : ((x xor y)= (if
((x= '1') xor (y = '1')) then '1' else
'0')).
```

### Proving The Design Correct

Penelope includes a simple proof editor/checker for predicate calculus that provides a number of proof rules for performing simplification and proofs. Penelope applies the rules according to user directions and indicates to the user what, if anything, still has to be proved after each step.

ORA would like Penelope to automatically simplify the logic conditions that it computes, putting them in the most convenient form, and to automatically prove the verification conditions if possible. Unfortunately, all but the most trivial simplification and proof in Penelope require the guidance and control of the user. This

interaction is necessary because of the well-known fact that simplification and theorem proving are in general undecidable; even so called automatic theorem provers usually require a good deal of guidance from human beings.

Each statement to be proved or simplified is presented in the form of a sequent, a set of hypotheses and a conclusion. In Penelope a sequent is displayed with numbered hypotheses and the conclusion is indicated by >>. Penelope displays the sequent (n>=0) => (0<= n) as follows:

```
--! 1. (n>=0)
--! >>(0<=n)
```

Note that all lines of a proof begin with the compound delimiter -- ! .

Each proof in Penelope takes place in the context of an available theory. Within a VHDL design unit, the theory is determined by entity declaration annotations and all the local lemmas currently being applied to complete the proof. The theory that is available for proving a given lemma consists of the axioms, assumptions, and proved lemmas that precede the given lemma.

Penelope's proofs are tree-structured. Each node of the tree corresponds to a sequent to be proved and one proof step that proves it, possibly subject to proving other, derivative sequents. For example, to prove the sequent  $\tau \Rightarrow a$  and  $b$ , you can use a rule we call and-synthesis, which commits you to prove instead the derivative sequents  $\tau \Rightarrow a$  and  $\tau \Rightarrow b$ . The children or subproofs of the node correspond to the further sequents needed to prove it. Leaves of a completed proof correspond to sequents that require no further proof (e.g., a sequent whose conclusion is "true"). While constructing a proof, the leaves also include unproved sequents.

The proof rules are organized with a hierarchical menu. When the cursor is positioned at a proof step, each item on the menu may represent a proof rule or a group of proof rules. For example, thinning is a proof rule, but analyze-hypothesis represents a group

of proof rules. If the help-pane menu item corresponds to a single proof rule, clicking on the menu item causes the proof rule to be added to the proof tree. If a group of proof rules is chosen, a submenu appears with the individual rules in the group.

The large number of proof rules available may make the Penelope prover seem formidable. Penelope's proof steps fall into several basic groups: the application of automatic simplifiers or rewriters; the application of some available theorem (called instantiation); rules (such as and-synthesis, mentioned above) that break down the conclusion or hypotheses according to their syntactical form; and proof-structuring rules, such as proof by cases or proof by induction.

### Verification Examples

This section provides two small examples of verification of VHDL designs. The examples are trivial but the information provided by the definition of VHDL semantics can be shown, and the verification condition that must be satisfied to meet the specification can be seen.

The two cases are shown in some detail. In each case, the designer always provides three things, an entity declaration, an architecture for the entity, and a specification written in Larch. The specification is just the port behavior the designer intends by virtue of his/her design.

The fourth component in each case is a replay of portions of the proof. This is done in some detail, but the very mundane steps are omitted.

The first case consists of a design with two signal assignment statements:  $c \leq \text{not } a$ ; and  $b \leq \text{not } c$ ; and the specification is  $b = a$ .

The VHDL entity/architecture description is:

```
entity simple2 is
port (a : bit;
```

```
        b : out bit);
end simple2;

architecture arch of simple2 is
signal c : bit;
begin
    c <= not a;
    b <= not c;
end arch;
```

The Larch specification is:

```
entity simple2
guarantees forall t :State::
    advance(t)>0 -> ((b@t)=(a@t))
end
```

The Verification Condition generated by the Larch/VHDL tool is:

```
b: -> Signal[Bit]
a: -> Signal[Bit]
c: -> Signal[Bit]

asserts
forall t:State
advance_lemma: forall t:State::
    advance(t)>0
    ->
    c() @ prev(t) = c()@la
    and
    a()@prev(t) = a()@t

implies
equations
vc: ((forall t:State::c()@t =
        ((not a())'delayed(0))@t)
    and
    tracked(0, not a()))
and
    ((forall t:State::b()@t =
        ((notc())'delayed(0))@t)
    and
    tracked(0, not c()))
->
(forall t:State::advance(t)>0->
    b()@t = a()@t)
```

With the following guide it is easy to read the Verification Condition (VC) and to have confidence that the VC is true, and provable.

- The signals a,b,c are identified as signal of type Bit.
- The advance lemma says t will not advance until a and c are stable.
- Verification theorems are of the form  $A \rightarrow B$ , with A derived from the architecture and B from the Larch specification. So the designer is

obligated to prove B, given A. In other word, prove the specification is satisfied, given that you have the behavior specified by the architecture.

- d. The A part has two parts, one for each of the two assignment statements in the architecture.

After transforming the VC using routine `measures`, we get:

```
--| proof:
{Routine steps deleted}
  >> not(not(a@t)) = a@t
```

This certainly seems provable. In order to get to this point, nothing but routine substitutions were used. It is clear we need to use the VHDL semantics for 'not' to complete the proof. The library of theories in `VHDL_MATH` has the necessary theorem. Instantiate it and do a routine simplification to get:

```
>> a@t = '1' or '0' = a@t
```

We are required to show that `a` is either '1' or '0'. But `a` is a signal of type `Bit`, and the `VHDL_MATH` library contains the needed theorem to complete the proof.

```
BY synthesis of TRUE
```

In the second case a 3 input and-component (`and3`) is instantiated twice to create a 5-input and gate.

Since a 3-input and gate is used as a component, the specification of the component is available. It is the specification of the 3-input and , rather than its architecture that will be used to prove the design of the 5 bit gate is correct.

The VHDL entity/ architecture is:

```
entity and5 is
  port(a,b,c,d,e : bit;
        z : out bit);
end and5;

architecture arch of and5 is
  component and3
    port (a,b,c: bit;z : outbit);
  end component;

  signal temp : bit;
```

```
begin
  L: and3 port map(a,b,c,temp);
  M: and3 port map(d,e,temp,z);
end arch;
```

The Larch specification is:

```
entity and5
  guarantees forall t :State::
    advance(t)>0 ->
      (z@t) =
        ((a@t) and (b@t) and (c@t)
         and (d@t) and (e@t)))
end
```

Here is the VC for the 5-input and-gate case.

```
z : -> Signal[Bit]
e : -> Signal[Bit]
d : -> Signal[Bit]
a : -> Signal[Bit]
b : -> Signal[Bit]
c : -> Signal[Bit]
temp : -> Signal[Bit]

implies
equations
vc: (((forall t:State::
      ((advance(t) > 0) ->
        ((temp()@t) = (((a()@t)
          and
            (b()@t))
          and
            (c()@λa))))))
and
(forall t:State::((advance(t) > 0)
->
((z()@t) = (((d()@t)
and
(e()@t) and (temp()@t)))))) ->
(forall t:State::((advance(t)>0)
->
((z()@t) = (((a()@t)
and
(b()@t) and (c()@t))
and
(d()@t) and (e()@t))))))
```

The form is interesting. There is no `advance_lemma` because there are no processes in the design, only instantiations. The VC is still of the form `architecture -> specification`.

It is easy to see the specification is just `z = a and b and c and d and e`.

The portion of the VC derived from the architecture has two parts, one for each instantiation of the 3-input and-

gate. So, the proof will require we show:

z = a and b and c and d and e  
given that

temp = a and b and c  
z = d and e and temp

This seems provable.

The proof after the usual processing is:

```
--| proof:
  1. advance( λa) > 0
  2. temp@λa = ((a@λa and b@λa)
    and c@λa)
  3. z@λa = ((d@λa and e@λa)
    and temp@λa)
>> z@λa
  = (((a@λa and b@λa) and
    c@λa) and d@λa) and e@λa)
<proof>
```

Only substitutions and simplifications are needed to complete the proof. There are three hypotheses, or given facts. From these we are required to prove the output z meets its specification.

Here we see the great potential for theorem proving methods for hierarchical design. The proof of correctness for the 5-input and gate had little to do with the semantics of VHDL process statements. Instead the specified properties of components were logically combined to prove correctness of a network of components.

## Conclusions

Larch/VHDL is an interactive environment that helps its user to develop and verify digital electronic hardware designs written in VHDL. The Larch/VHDL environment provides a means to specify and verify a hardware design prior to simulation and in a manner that supports specification and design reuse. Of course, Larch/VHDL can also be used to verify previously written VHDL models by developing the Larch

specification for the models. With the theorem Penelope, it is possible to modify a previously verified model and verify the new model with minimal effort by replaying and modifying the original model's proof.

In summary, Larch/VHDL is the user's trained assistant in verification. It performs well-defined but tedious tasks, like computing verification conditions and carrying out proof steps, while the user is responsible for the intelligent part of the work, specifying the design, developing the design, and deciding how to prove it.

## References

- [1] Edsger W. Dijkstra, A Discipline of Programming. Prentice Hall, Englewood Cliffs, 1976.
- [2] S. Garland, J. Guttag and J. Horning, "Debugging Larch Shared Language Specification," IEEE Trans. on Software Engineering, Vol. 16, no. 9, September 1990.
- [3] David Gries, The Science of Programming. Springer-Verlag, 1981.
- [4] J. Guttag, J. Horning and J. Wing, "The Larch Family of Specification Languages," IEEE Software, September 1985.
- [5] D. Jamsek and M. Bickford, "Formal Verification of VHDL Models," Final Technical Report, Rome Laboratory RL-TR-94-3, March 1994.
- [6] J. Wing, "Writing Larch Interface Language Specifications," ACM Trans. on Programming Languages and Systems, Vol. 9, no.1, January 1987.
- [7] Odyssey Research Associates, "Penelope Reference Manual V3-3," TM94-0009, December 1993.