

## A C++ Model of VHDL for the Transformation of Parallel Algorithms

Robert E. Anderson  
VLSI Technology, Inc.  
Ste 214, 10250 SW Greenburg Rd.  
Portland OR 97223  
(rob.anderson@vlsi.com)

**Abstract-** The introduction of new ASIC technologies every few months begs the question of how we are to ensure the optimal mapping of algorithms to hardware. VHDL is not suitable for general prototyping at this level, but the modelling concepts in VHDL are valuable. A C++ model of VHDL can keep the relevant concepts, and provide a richer environment for dynamic prototyping.

This paper describes the current state of a C++ prototyping system done at VLSI. The paper also discusses VHDL "missing features" which make it necessary to use C++ and what these features would mean in VHDL.

**Introduction-** In standard cell design, we are interested in parallel algorithms such as multipliers and adders. These characteristically have high gate counts (5 to 20k gates) and speed requirements in the sub-10 ns region for CMOS.

General synthesis is inappropriate for this purpose. For high performance it is necessary to experiment with new gate types, new kinds of interconnect, new algorithms, and custom blocks. It is not feasible to use commercial synthesizers for this.

C++ classes were written to form a prototyping system. The C++ classes reflect the elaborated model of Wilsey (<http://www.ece.uc.edu/~paw/rassp.html>). In addition there are algorithm classes to encapsulate parallel algorithms.

**The VHDL Classes-** Current VHDL classes include:

```
vtype(name,parent_name,values,index range,res_fcn)
port_assoc(actual,formal_fcn,actual_fcn)
port(name,mode,type)
signal(name,type,kind)
component_inst(cdecl,name,port_assoc_list)
```

```
csig_assign(dest,wfsig)
entity(name,ports)
architecture(name,signals,statements,comp_decl,subp
rogs)
component_def(name,types,ports,entity,architecture)
```

In the above, `vtype` includes type and subtype concepts; `component_def` is an encapsulation of the entity and architecture which could be contained in another `component_def`. Thus, the current classes are sufficient to model structural designs with hierarchy.

This approach allows for other parts of VHDL to be added easily. All of the objects inherit characteristics from base classes which have polymorphic methods for file I/O and dumping (for debugging). In addition there are collections with sort and search capabilities.

With C++ it was easy to add the ability for a model to write itself in VHDL. In concept, an algorithm creates a tree structure of VHDL components. The top object is a `component_def`. Any `component_def` object can write itself out in VHDL. The VHDL can be simulated or synthesized.

**C++ compared to VHDL-** The principal advantage for C++ is dynamic instantiation. In C++ objects are created and deleted by the NEW and DELETE operators. Furthermore, the objects are handled by pointers so they may be used in other dynamic structures.

VHDL has no run-time facility to instantiate components or signals, or create and add new drivers to signals. There are access types in VHDL, but these do not point at component instances: the conceptual objects of VHDL.

The principal advantage of VHDL is the simulation engine which allows the modelling of real time sys-

tems. Also, there are simulators and synthesizers for VHDL. C++ is too flexible to be of use in either of these areas.

Another advantage of VHDL is that it is easier to read than C++. VHDL has rather rigid semantics in comparison to C++. It is easier to come back to a VHDL design after some time, and understand the model quickly.

If VHDL had these C++ facilities, it would be an object oriented language. The language would then be capable of modelling time variant behavior. An example of time variant behavior is reconfigurable logic. Without these object-oriented facilities, such modelling becomes very structural.

**Implementing an Algorithm-** Each algorithm class has a method: `component_def* algo_name::create_algo()`. The method is invoked with size or other parameters, and a component is returned with the algorithm implemented.

The top level component contains type definitions for ports and internal signals, an entity definition, and an architecture.

Within the architecture is a list of component declarations, each containing a list of ports. There is also a list of signals, and a list of statements.

Each algorithm is different, but the Wallace Tree is a good example of an algorithm which can be experimented with. The following example shows the prototyping strength of this C++/VHDL system.

**Wallace Tree Creation-** The following procedure was implemented to cover the Wallace Tree algorithm:

- 1) create N+M lists for signals, corresponding to output sums
- 2) create 'and' gates and place the output signals in these lists
- 3) traverse the lists in order; for each list:
- 4) fit compressors to the list until there are only two remaining signals; move carries to the next list.
- 5) Call another algorithm to create an adder component.
- 6) Finished

In the implementation phase, we did a lot of experimenting with different kinds of compressors. In this system, such changes require the addition of another component declaration, then a change to

the step#4 fitting algorithm. These changes were easy to make.

One idea which paid off was to encapsulate the fitting criteria for components in the component instance objects themselves. Basically, timing arcs are known by the component, and there is information in each signal about the cumulative delay. During the process of instantiation, the component itself makes decisions about connection order.

**Testbench Generation-** The algorithm classes also have procedures to generate VHDL code for testbenches. The testbenches use the IEEE `numeric_std` package.

**Production Program Generation-** The algorithm classes have procedures to generate final programs for use in generating the VHDL and script files to implement parallel algorithms in standard cell areas.

**Future Plans-** We may experiment in embedding positional information in the models, to aid standard cell placement.

**Conclusions-** There is an advantage to modelling VHDL constructs as classes in C++. Such a C++ class library is useful in prototyping parallel algorithms, and keeping track of ideas during this work.

VHDL could be augmented with several facilities like those in C++. The result would be a more object-oriented VHDL with the capability to model time variant behavior.

It would be desirable to use VHDL for this kind of work, and keep everything in one language. It is very costly in terms of man-hours to adopt both languages and continuously switch between them.