Semi-Automated Validation of VHDL & Related Languages

John Willis
FTL Systems, Inc.
Rochester, MN 55906
jwillis@ftisys.com

Gregory D. Peterson
Wright Laboratory
Wright Patterson AFB
Dayton, Ohio
gdp@aa.wpafb.af.mil

Philip A. Wilsey
University of Cincinnati
Cincinnati, Ohio
paw@thor.ece.uc.edu

John Hines
Wright Laboratory
Wright Patterson AFB
Dayton, Ohio
hines@el.wpafb.af.mil

William H. Dashiel
National Institute of Science and Technology
Gaithersburg, Maryland
dashiel@speckle.ncsl.nist.gov

Abstract
Comprehensive and affordable evaluation of the fidelity with which a VHDL tool implementation and VHDL language specification correspond is the focus of this research. The research is embodied in a practical semi-automated validation tool, VIVA.

Previous work in this area focused on hundreds to thousands of manually written test cases (and potentially expected result files). Such suites were expensive to develop, expensive to maintain, and often conferred less than optimal language coverage due to practical limitations on the available development resources.

Research and development efforts described in this paper utilize several, comparatively terse adaptations of a language specification to drive a semi-automatic test suite generator and eventually a VHDL tool under test.

Specification files derived from the language specification express lexical, syntactic, semantic and temporal properties of a concurrent, imperative language. The language being specified may include both intrinsic functionality (such as IEEE Std. 1076-93 [1]) and extrinsic functionality (such as the standard logic packages defined by IEEE Std. 1164 [2]).

Manually supplied tuning parameters guide the validation suite generator as it emits a stream of VHDL test files, expected results and derivative information used to facilitate human analysis of failed tests. Tests evaluate both a tool’s ability to process correct input and a tool’s ability to discriminate errors in the input stream.

Whereas VIVA’s primary focus is validation of tools implementing VHDL-related standards, preliminary analysis suggests that the techniques are extensible to support validation of other concurrent, imperative languages such as Ada. Temporal validation of constraint-based languages, such as the proposed VHDL-AMS extensions to VHDL, may require further work.

The Challenge
Migration of designs and designers among tools motivates hardware description language (HDL) standards and thus evaluation of a specific tool’s compliance with the language standard. In the absence of compliance evaluation, designs and designers will require substantial effort (and expense) in order to utilize a variety of HDL tools or, worse, designs are more likely to be fabricated and deployed with errors. Without a comprehensive evaluation of a tool’s fidelity to
applicable HDL standards, the promise of HDL standards remains unrealized.


**Previous Work**

Previous approaches to HDL language validation make intensive use of human effort, thus are very expensive to create, expensive to maintain, and detect a limited number of errors in VHDL tool implementations.

Large validation suites have been manually developed by MCC (VHDL Version 7.2 updated to 1076-87), Intermetrics (IEEE Std. 1076-87) and most recently The VHDL Technology Group (1076-87). These validation suites provided and continue to provide a critical resource for VHDL tool developers, but is this methodology ideal?

The latest such validation effort [4] methodically analyzed the IEEE Std. 1076-87 language reference manual to guide authoring of almost 3,200 test cases. The authors estimated that at least an additional 10,000 test cases are needed to extend this methodology to embrace just the base VHDL language standard.

Extrapolating manual test generation to embrace VHDL-related standards now in use or anticipated over the next 3 years suggests 20,000 to 70,000 new test cases are needed. Based on writing one test case every person / hour, such a comprehensive effort would require 10 to 40 person - years and cost several million dollars. With new language revisions every five years, a team of several dozen skilled VHDL programmers, test-case architects and support staff would be required for timely validation suite development.

The cost and delay in initially developing and maintaining manual language validation suites strongly motivates a search for new and less manually intensive approaches. This paper describes one such approach to provide the VHDL user community with comprehensive and economical validation.

**Figure 1. Semi-automated approach to suite generation and validation of tool under test.**

**Approach**

VIVA introduces a new, automated approach to generation, application and analysis of validation tests. Figure 1 schematically illustrates the new approach.

In this approach there are five critical software components. A set of terse language specifications, derived from the applicable language standards, drives the validation process. A set of configuration parameters define heuristic aspects of a particular validation run, such as the number of tests to be applied, test emphasis and hardware resources available for testing. The validation suite generator combines language specifications and configuration parameters to generate, interactively or off-line, a stream of test cases, expected results and linkages back to the applicable standard. A wrapper applies the stream of validation test cases to the tool under test and monitors output from the tool under test, summarizing the validation results. Finally the tool under test may be a language analyzer, simulator, synthesis compiler / hardware emulator or any other tool (suite) capable of manifesting behavior defined by a VHDL language standard.

Once language specifications and configuration parameters have been prepared for the validation suite generator, it becomes feasible to generate tens to hundreds of tests per second, 24-hours per day, 7 days per week. Nearly six million tests may be generated per week, potentially resulting in a degree of validation coverage not
approachable by previous, manual validation methods.

Insuring that each test (out of millions) provides substantially new validation information as well as latency through the wrapper / tool under test limit the number of validation tests which can actually be completed. Our current effort effectively addresses both limitations. Whereas previous validation suites have concentrated on analysis time tests of correct VHDL syntax and first-order faulty VHDL syntax, additional test generation capacity enables expanded testing, including: lexical validation, analysis / elaboration / runtime semantics, temporal semantics, stress testing, and nth order failure analysis. All of these additional test modalities provide information on the fidelity of language implementation which is relevant to real-world use and miss-use of VHDL tools. These additional validation modalities can easily result in many millions of tests.

Once we can generate hundreds of thousands to millions of tests, latency through the wrapper and tool under test becomes the limiting factor restricting test coverage. Parallel application of validation tests can increase bandwidth through the wrapper / tool under test by factors of ten to hundreds. Figure 2. illustrates such a parallel approach to validation testing.

Our approach divides validation into seven different kinds of tests: lexical, syntactic, analysis-time semantics, elaboration-time semantics, liveliness, functional, and temporal. The following sections describe each of these tests and their specification in more detail.

The breadth of validation tests provided by VIVA require a variety of specification files. VIVA's lexical, syntactic and semantic language specifications are derived from the DLG and ANTLR parser specifications provided by the Purdue Compiler Construction Tool Set (PCCTS) [5]. Functional specifications are derived directly from the text of packages and package bodies. Temporal specifications use declarators and structure introduced by syntactic specifications along with temporal logic assertions.

**Lexical Validation**

Lexical validation evaluates a tool's ability to properly segment VHDL source files into a stream of lexical tokens. Examples of lexical tokens include keywords, identifiers and decimal literals.

The lexical validation component of the validation generator emits a stream of lexical tokens in response to calls from syntactic and higher-level test generation. Streams of such lexical tokens are then segmented to form a stream of test cases.

VIVA's use a specification file derived from PCCTS's DLG lexical token specification. Each lexical token is described by one or more token directives, following the examples shown in Figure 3.

In VHDL, keywords such as `WITH`, appear in the test case stream as a potentially mixed case sequence containing a W, an I, a T and an H. When generating a test case intended to form correct VHDL, lexical validation permutes the valid combinations of the lexical rule to emit identical files except for permutations such as "WITH", "WITH", "With", and "with". Tests of a tool's ability to detect incorrect VHDL require the lexical validation component to emit a variety of tests which emit character sequences excluded from the lexical rule in question and...
any other lexical rule permitted at a given point in the syntactic generation process.

Figure 3. Example of a DLG lexical rule specifying a VHDL decimal literal.

```
#token WITH "[Ww][Ii][Tt][Hh]" <<...>>

#token IDENTIFIER
[ConstraintSet c] >
"[a-zA-Z][a-zA-Z0-9]*" <<...>>

#token DECIMAL_LITERAL
[ConstraintSet c] >
"[0-9][0-9]* { [0-9][0-9]* }
{(E{[+\-]} [0-9][0-9_] )}" <<...>>
```

Lexical tokens such as an identifier and decimal literal are conceptually more complex in that they have many more permutations than can be readily generated. Taking into account constraints which are passed into the generation action (contained within the elided << and >>), either a minimal character sequence meeting the constraints of a more complex variant may be emitted. For example, a semantic rule may propagate through the syntactic component requiring a decimal literal greater than or equal to zero and less than the largest positive integer which can be represented in 32 bits.

**Syntactic Validation**

Syntactic validation evaluates a tool’s ability to accept any legitimate sequence of valid lexical tokens and to reject any illegal sequence. As a practical matter, any syntactic validation tests should also have legal (all though not comprehensive) semantics (see below). Examples of syntactic productions include concurrent BLOCK statements and logical operators.

VIVA uses a syntactic production specification derived from PCCTS’s ANTLR parser specification. ANTLR’s generalized parameter passing mechanism serves to pass semantic constraints (see next sections) downward in the production hierarchy so as to govern the generation of lexical tokens (described in the previous section). PCCTS’s parser specification does not pass parameters to lexical productions and return a single parameter, VIVA requires a more general, consistent mechanism for both terminal (lexical) and non-terminal (syntactic) productions.

Figure 4 illustrates the syntactic production representing concurrent block statements. This production includes references to lexical productions including BLOCK, IS, BEGIN, and END. Other lexical productions are indirectly reachable through the non-terminal productions block_label, optional_guard_expression, block_header, block_declarative_part, block_statement_part, and optional_block_label. The two elided constraint actions (<< ... >>) include code needed to enter and exit (respectively) the declarative region represented by the block statement.

The block_label (non-terminal) syntactic production is shown in the lower part of Figure 4. This production consists of a single call to a terminal (lexical) production with a constraint indicating that this identifier must be a unique block label within the declarative region.

Figure 4. Example of syntactic production for a concurrent block statement and its label.

```
block_statement [IIR* region]
 : << IIR* label; >>
 block_label > [label] "::"
 BLOCK
 << .. >>
 optional_guard_expression
 { IS }
 block_header
 block_declarative_part
 BEGIN
 block_statement_part
 END
 BLOCK
 optional_block_label
 "::" << .. >>
 ;

 block_label > [IIR* label]
 :
 IDENTIFIER
 [IsUniqueBlockLabel]
 << ... >>
 ;
```
Expressions require a more complex constraint environment than the concurrent block example in Figure 4. Figure 5 illustrates the OrOperator syntactic production including implementations which are resolved to overloads of the OR operators.

The OR operator production (shown in Figure 5) is called with constraints specifying the type (base or constrained subtype) of the resulting expression, a constraint denoting locally static, globally static or runtime evaluation of the expression, and a kind constraint. For example, the kind constraint might denote expressions consisting of a signal which could appear on the left-hand-side of a signal assignment. Intermediate representation code representing the value being constructed returns from these productions.

Several constraint actions are elided in Figure 5. The first and third elided constraint action controls the number of repetitions through the OR, expr7 production will execute. The second elided constraint action uses a heuristic to choose an intrinsic OR operator or an OR operator previously defined as a subprogram within this test case. The fourth elided constraint action completes definition of the resulting expression, returning a value.

Whereas lexical and semantic validation tests should have legal semantics, such tests are not intended to providing a spanning validation of all legal semantic constraints. Such a comprehensive semantic validation may be segmented into semantic constraints which may be tested during analysis (analysis-time semantic validation), during elaboration (elaboration-time semantic validation), during execution of a single delta cycle or “instant” of time (functional validation) and between instants of time (temporal validation). The next four sections briefly describe these tests and how they may be automatically generated from terse specifications prepared by a human.

**Analysis-Time Semantic Validation**

Analysis-time semantics consist of language-mandated constraints which are generally described in language reference manuals by prose and not grammar. Examples include insuring that a process appearing directly within an entity is passive, that a sequential exit statement refers to a loop label, that an expression is locally static, or that an expression has a unique interpretation at appropriate points in the syntactic production.

In order to generate a set of validation tests that approach a comprehensive exploration of legitimate and illegal semantics, both the constraint classes and constraint actions must be enhanced so that a sequence of automatically generated tests comprehensively samples the allowable and explicitly disallowed semantics (rather than merely insuring that the emitted tests are semantically valid). For both analysis-time and elaboration-time semantic validation the same PCCTS-derived language specification introduced above appears sufficient.

Exhaustive testing of allowed and explicitly disallowed semantics is computationally infeasible. Such an effort is roughly like thousands of monkeys exhaustively writing and testing all semantically allowed and disallowed VHDL designs. Practical semantic validation test generation requires a breadth-first traversal of the
production rules, heuristic pruning, and heuristic choice of parameters.

Breadth-first traversal of the productions means that the validation generator would perhaps generate a set of tests with a signal in a given declarative region, then a variable and finally a test with a constant rather then generating a test cases with a second signal, a third signal, a fourth signal and so on.

Heuristic pruning carefully chooses parametric values rather than attempting exhaustive enumeration. For example, the condition of an IF expression must be of type boolean (analysis-time semantic constraint). Early in the validation of semantically rejected constructs it is better to insure that the analyzer rejects an integer, a real, other enumerated types, physical types, arrays, records, and access types by trying a couple of varied examples of each rather than attempting validation with a broader range of non-boolean types.

Heuristic choice of parameters tries to maximize the chance of detecting an error as early in the test sequence as possible rather than trying to distribute the probability of error detection equally among all test cases. Examples of such heuristics expedite validation of operator precedence, aliases, and visibility rules.

### Elaboration-Time Semantic Validation

Many semantic constraints apply during or after elaboration. For example, elaborated objects must have a constrained type; some globally-static expressions must be suitable for evaluation during or immediately following elaboration; and some interfaces require type-compatible bindings of actual values to formal parameters.

Since test cases are generated as a sequence of zero or more, un-elaborated design units, application of some elaboration-time semantic rules to the validation generation process is more complex than analysis-time semantic checks. In particular, greater underlying state information is required during validation generation and constraint actions must take these underlying state into account.

### Liveliness Validation

Liveliness validation establishes a tool’s ability to interact with its environment through assert statements, report statements and textio subprograms. Correct implementation of such functions are critical to subsequent functional and temporal validation.

These validation tests are most readily constructed by permuting a small number of test case templates across a variety of data types and output sequences in order to generate VHDL source files and expected result files.

### Functional Validation

Functional validation verifies that expressions and side effects visible within the same instant of time comply with language-defined semantics. For example, an addition expression should implement the language-defined addition operator; an assignment should result in language-compliant assignment; and subprogram calls should perform language-compliant dynamic elaboration.

Functional validation involves exploring a tool’s implementation of both intrinsic VHDL functionality (such as the addition of two integers) and functionality defined in terms of intrinsic VHDL functionality (such as the logical OR of two values of standard logic type). Intrinsic and extrinsic validation requires different test case generation techniques; validation of intrinsic functionality is a precursor to validation of extrinsic functionality.

In order to de-couple distinct functional tests it is useful to run each test within a process having the minimal context needed to properly setup and evaluate the expression or sequential statement(s) under test. A variety of statements and expressions may readily be constructed from the grammar and semantic specifications derived above. Heuristics introduced in the functional validation phase attempt to maximize the probability of identifying tool errors early in the validation test sequence through breadth-first traversal of the grammar and judicious choice of values for expression terminals which expose boundary cases in the expression evaluation.

Once functional test cases have been formulated, reliably acquiring expected results is non-trivial.
A validation process should not assume the correctness of any "golden simulator", compiler or even processor instruction set implementation. Thus expected results for functional tests must be pre-computed and stored. Pre-computed results may derive by running functional test cases on a variety of HDL implementations, comparing results until consensus is achieved.

Through its long-standing validation activities, the National Institute of Science and Technology (NIST) has assembled a substantial collection of functional validation tests for languages other than VHDL. Since the expected results from such tests have been established over an even broader range of implementations than would be possible for VHDL alone, there is substantial incentive to adopt other NIST functional validation suites where feasible.

Functional validation of VHDL-related standards defined in terms of package bodies depends on prior validation of intrinsic functionality and availability of completely defined package bodies. For example, validation of the math package via automatic test generation requires availability of a “correct” package body implementing all of the mathematical functions. Static analysis must establish a set of correct results expected for each case. High-fidelity validation again requires that consensus on the expected result be achieved through consensus among implementations.

Following functional validation of expression evaluation and sequential statement execution, the challenge of validating the correct interleaving of side-effects between VHDL processes remains. We refer to this final validation step as temporal validation.

**Temporal Validation**

Temporal validation insures that two or more processes, communicating via changes in signal and/or shared variable values, commit and reference side effects with a permissible partial ordering. Temporal validation must deal with the complexity resulting from numerous correct expected results.

By relying on prior functional verification, verification may assume that the value being written by an individual assignment is correct. Thus temporal verification concentrates on exploring both the order in which values assigned to signals and shared variables become visible to other processes and the value transformations resulting from overlapping assignments.

Partial ordering of assignment and reference to a shared object may occur within a single delta cycle (shared variables), between delta cycles, and between a interval within VHDL’s finite time domain.

Various forms of temporal logics may be used to specify ordering relationships using language structure already specified by PCCS-derived specification files (see above). The evaluation generator then uses a supply of unique marker values to construct specific instances of assignment and reference permitted (or disallowed) by the temporal logic rules. Such test cases are the temporal analog of syntactic and semantic test cases derived from grammars during earlier validation phases.

Value transformations result from use of inertial delay mechanisms on a single driver and resolution functions transforming two or more concurrent drivers for the same signal. Under such transformations, values assigned to a shared object (signal or variable) may never become visible to any other process or may take on a new value before becoming visible to another process.

Temporal validation in the presence of value transformations is slightly more complex. Rather than simply being unique, marker values chosen for this later form of temporal validation must remain unique and identifiable through the specified value transformations such that expected results may trace a value from assignment to reference. There is mathematical precedent for such marker values and transformations in the design of self-checking arithmetic logic units (in which check bits are carried through arithmetic operations for subsequent integrity verification) [6].

Temporal validation techniques complete a suite of HDL tool verification techniques believed to provide a higher degree of verification fidelity than would be possible with any feasible manual test cases generation approach. While providing arbitrarily high degrees of verification fidelity, generally at the cost of increased computing.
time, no such techniques are known which can
provably identify all possible deviations between
language specification and tool implementation.

**Impact**

The authors intend that results of this work be
widely available for the evaluation of any tool
intended to support VHDL. It is expected that
binary forms of the tool will be available so that
the generator can be run interactively on several
widely available, platforms (including x86
computers running Windows NT or Linux and
SPARC processors running SunOS/Solaris). The
remote wrapper for tools under test enables
testing of almost any tool/platform combination
with remote-access capability (such as Ethernet
or even serial/parallel-port/tape communication).

Whereas software patents are likely to be issue on
aspects of this project (patents-pending
disclaimer), it is the authors' intent that VIVA
releases be freely available via WWW and
software publication channels with rights to
freely use and redistribute VIVA for the
validation of any VHDL-related tools (US
Government agencies have broader rights). The
authors reserve the right to provide replacement
software via WWW with generally comparable
functionality in the event that VIVA software is
found to violate any other party's valid patent
claims. No warranty or license is implied or
expressed by FTL Systems, Inc., The University
of Cincinnati, or the US Government.

Anticipated WWW sites for information on this
project include:

- http://www.vhdl.org/vi/validation/
- http://www.ftlsystems.com/research/
- http://www.ececs.uc.edu/~paw/viva/

**Acknowledgments**

Wright Laboratory (USAF) is supporting this
research and development effort under contract
F33615-96-C-1909 (co-monitored by Wright
Laboratory and NIST). DARPA supported
research, under contract DABT63-96-C-0004,
leading to the current effort. This paper does not
necessarily reflect the position or policy of the
US Government; no official endorsement should
be inferred.

**Bibliography**

Manual, IEEE Std. 1076, The Institute of
Electrical and Electronic Engineers, New York,

for VHDL Model Interoperability
(Std_logic_1164), IEEE Std. 1164, The Institute
of Electrical and Electronic Engineers, New

[3] Federal Information Standards Publication:
VHSIC Hardware Description Language
(VHDL), FIPS PUB 172-92, June 29, 1992.

John W. Hines and Bill Billowitch. In
Proceedings of the Spring 1995 VHDL
International User's Conference, Conference

[5] Language Translation Using PCCTS and C++

[6] Microprogrammed Control and Reliable
Design of Small Computers. George D. Kraft and