AFRL-IF-WP-TR-2001-1531

VERY HIGH SPEED INTEGRATED CIRCUITS (VHISC) HARDWARE DESCRIPTION LANGUAGE (VHDL) INTERACTIVE VALIDATION ALCHEMY (VIVA)

Technology and Software for Semiautomated, High Fidelity Validation of VHDL-Related Tools

Patrick Gallagher, Robert Newshutz, Sathyanarayanan Seshadri, Senjeev Thiyagarajan, and John Willis

FTL Systems, Inc.
1620 Greenview Dr. SW
Rochester, MN 55902-1034

MAY 2001

FINAL REPORT FOR 09 JULY 1996 – 31 MAY 2001

Approved for public release; distribution is unlimited.

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

Report Documentation Page

Report Date	Report Type	Dates Covered (from... to)
01MAY2001	N/A	09JUL1996 - 31MAY2001

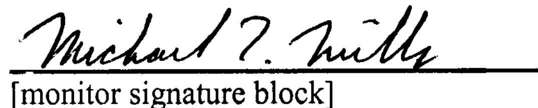
Title and Subtitle Very High Speed Integrated Circuits (VHISC) Hardware Description Language (VHDL) Interactive Validation Alchemy (VIVA). Technology and Software for Semiautomated, High Fidelity Validation of VHDL-Related Tools	**Contract Number** **Grant Number** **Program Element Number**
Author(s) Gallagher, Patrick; Newshutz, Robert; Seshadri, Sathyanarayanan; Thiyagarajan, Senjeev; Willis, John	**Project Number** **Task Number** **Work Unit Number**
Performing Organization Name(s) and Address(es) FTL Systems, Inc. 1620 Greenview Drive., S.W. Rochester, MN 55902-1034	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB, OH 45433-7334	**Sponsor/Monitor's Acronym(s)** **Sponsor/Monitor's Report Number(s)**
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes The original document contains color images.	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** SAR
Number of Pages 68	

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

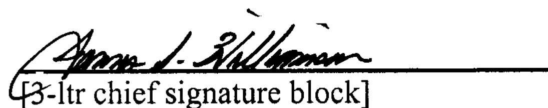
THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION



[monitor signature block]

[supervisor signature block]



[3-ltr chief signature block]

This report is published in the interest of scientific and technical information exchange and does not constitute approval or disapproval of its ideas or findings.

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
May 2001	Final	07/09/1996 – 05/31/2001

4. TITLE AND SUBTITLE

VERY HIGH SPEED INTEGRATED CIRCUITS (VHISC) HARDWARE DESCRIPTION LANGUAGE (VHDL) INTERACTIVE VALIDATION ALCHEMY (VIVA)

Technology and Software for Semiautomated, High Fidelity Validation of VHDL-Related Tools

5a. CONTRACT NUMBER F33615-96-C-1909

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER 65502F

6. AUTHOR(S)

Patrick Gallagher, Robert Newshutz, Sathyanarayanan Seshadri, Senjeev Thiyagarajan, and John Willis

5d. PROJECT NUMBER 6096

5e. TASK NUMBER 40

5f. WORK UNIT NUMBER 31

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

FTL Systems, Inc.
1620 Greenview Dr. SW
Rochester, MN 55902-1034

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

10. SPONSORING/MONITORING AGENCY ACRONYM(S) AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2001-1531

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT *(Maximum 200 Words)*

The objective of the VIVA program was to develop a tool to generate a suite of tests to validate the compliance of Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) tools to the standard definition. The test suite is semiautomated to enable maximum flexibility and coverage of the language definition, thus, precluding the introduction of language compliance errors in DoD systems designs that utilize VHDL. The approach includes lexical, syntactic semantic (analysis-time and elaboration-time), functional, and temporal tests. The test suite will include contextual situations and capacity testing in an interactive generation, test and analysis environment, for validating tools. The validation test generation tool development provides a lower cost, more reliable, and maintainable means for DoD/NIST to certify tools as VHDL-compliant. The approach also shows promise for helping automate other NIST certification tasks in the future. The tools can also be made available to VHDL vendors who want to test their newly developed tools for VHDL compliance so they can provide higher quality products to their customers.

15. SUBJECT TERMS
VHDL, Validation tools

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Michael T. Mills
Unclassified	Unclassified	Unclassified	SAR	72	**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-6548 x3583

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

VIVA Final Report
USAF Contract Number F33615-96-C-1909 1

CHAPTER 1 *Implementation Overview* **5**

System Architecture **6**
General Approach to Test Generation **13**
Test Generators for Lexical, Syntactic and Pre-Execution Semantics **15**
Test Generators for Liveliness **16**
Test Generators for Sequential Functionality **17**
Test Generators for Concurrent Functionality **18**
General Approach to Test Administration **19**
Organization of Subsequent Chapters **21**

CHAPTER 2 *Interface Common to All Test Generators* **23**

Interface Architecture **24**
Command Line Control Flags **26**
Command Stream Control Flags **28**
Status Stream Control Flags **30**
File Output Requirements **32**
Fault Tolerance Requirements **34**
Test Generator Message Format **36**

CHAPTER 3 *Test Generator Language Specifications* **37**

Introduction to Lexical and Syntactic Specifications **37**
Lexical and Syntactic Specification Grammar **39**
- *Lexemes* ***39***
- *Grammar* ***41***

Specification Constraints **49**
- *Strategy Constraints* ***50***
- *Kind Constraints* ***51***
- *Type Constraints* ***52***
- *Value Constraints* ***52***

CHAPTER 4 *Sequential Extrinsic Functional Test Generation* **53**

Introduction **54**

Type System **55**

Ordinal value selection ***55***

Arithmetic value selection ***56***

Composite value selection ***56***

Objects **57**

Subprograms--operations, functions, and procedures **58**

Processes and Concurrent **59**

Alternate Forms ***59***

Waveforms ***59***

Weaknesses **60**

Deliverables **61**

Future **62**

VHDL-AMS ***62***

FTG Rules **63**

FTG program structure ***63***

FTG rule creation ***63***

CHAPTER 5 *VIVA Installation Guide* **65**

CHAPTER 6 *Annotated Bibliography* **67**

CHAPTER 1

Implementation Overview

VIVA[1] is a system for generating and administering validation of a VHDL-related tool (or tool suite). Validation measures compliance between one or more VHDL-related standards and the tool under test. Applicable tools include not only source analyzers, but also simulators, synthesis tools, formal verification tools and related tools.

VIVA validates based on configuration and run-time inputs in order to yield run-time output. Configuration inputs to VIVA include an annotated language grammar (describing VHDL), constraints (semantic, sequential, concurrent), test strategies, and VHDL source fully specifying the functionality of VHDL-related libraries (such as standard logic). Run-time inputs adjust the specific validation regime and resources on which to conduct the validation. Outputs include a summary of failed tests and optionally the full annotated stimulus and response for any failed tests. Test file annotations provide a derivation from configuration inputs to the text of the test.

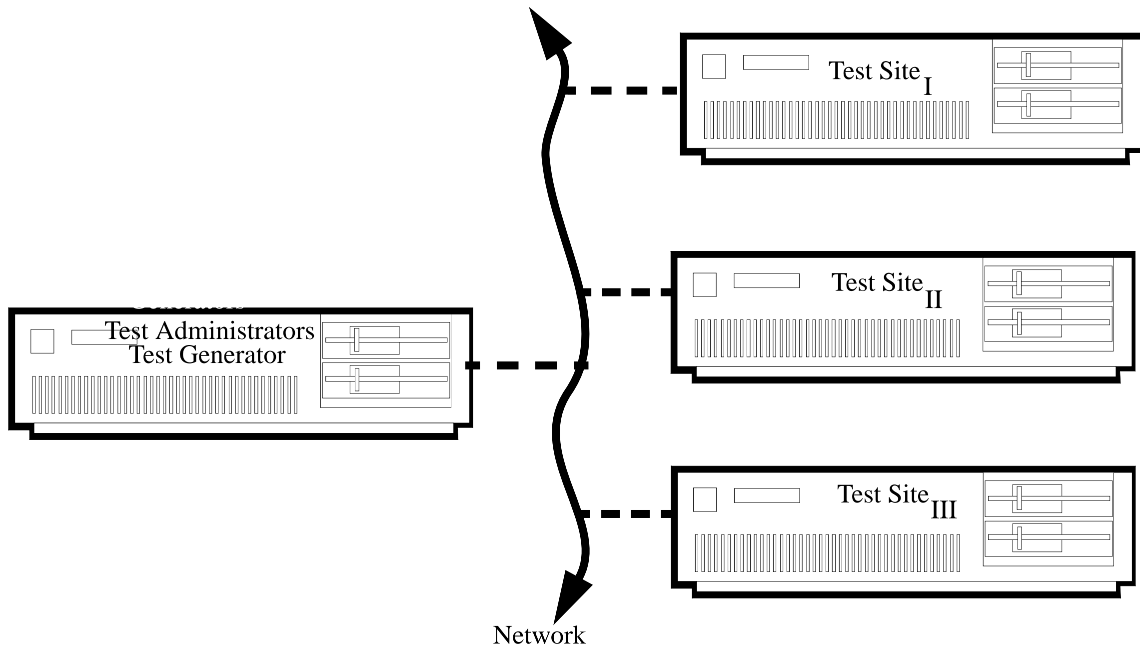
This chapter provides an overview of VIVA architecture and interfaces. Sections in this chapter describe VIVA's system architecture, general approach to test generators, specifics four of several distinct kinds of test generators (one per section), general approach to test administration, and the organization of subsequent chapters.

1. **V**HDL **I**nteractive **V**alidation **A**lchemy (name developed by Philip A. Wilsey).

System Architecture

The validation system architecture components involve a test administrator, one or more test generators and one or more test sites associated with a particular test generator. Figure 1 on page 6 illustrates a typical configuration of the system architecture in which both the test administrator and the test generator are located on a single computer. This computer corresponds via a TCP/IP network with three test sites (one running on each of three computers).

FIGURE 1. Example configuration of validation system architecture.

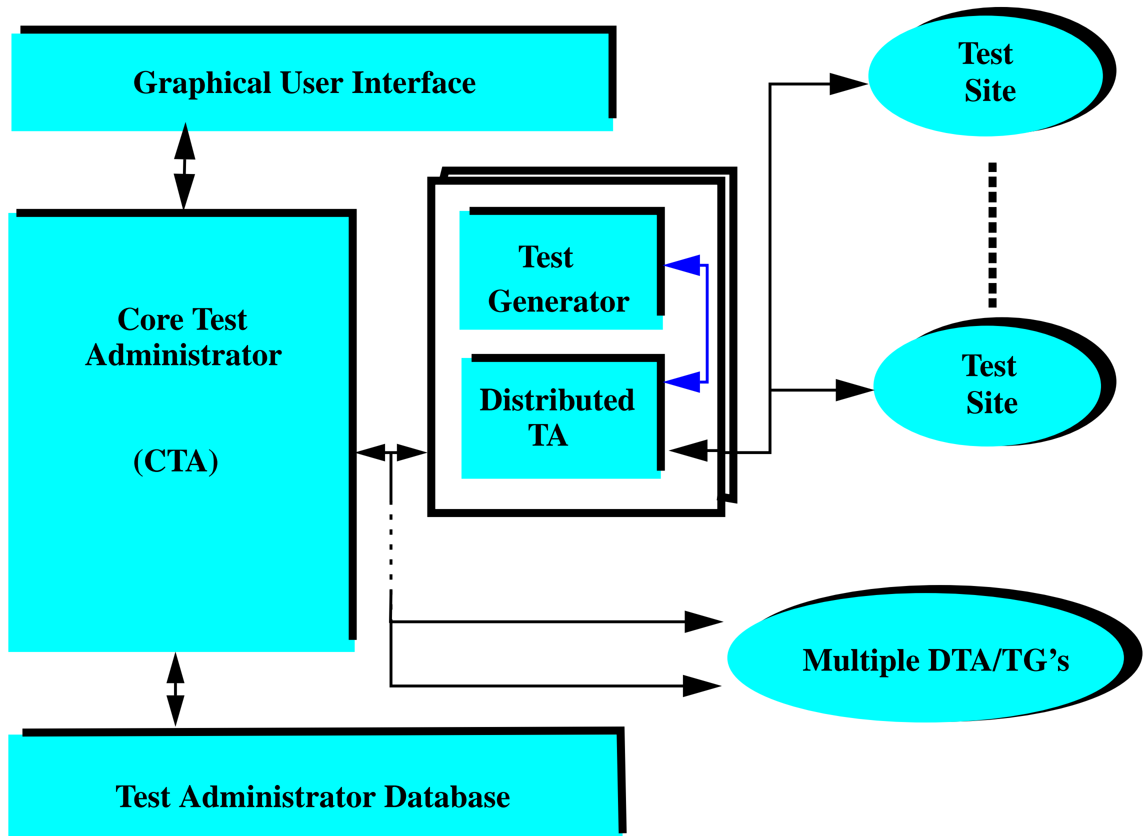


A variety of configurations other than the one shown above are possible. For smaller configurations, all components may run on a single, common computer as distinct operating system processes. For larger configurations, perhaps involving tens of computers, key components will be distributed onto one or more distinct computers. This architecture productively extends to several tens of computers, at which time the test administrator may also run on more than one computer. Substantially greater network bandwidth may also be needed that will provided more than that typically offered with TCP/IP over an Ethernet.[1]

1. Functionality developed under Wright Laboratory Contract F33615-96-C-1909 targets configurations with multiple test site computers and multiple generators are supported, however the core administrator runs on a single computer.

Figure 2 on page 7 illustrates the test administrator's architecture. Here the test administrator can be seen as two distinct blocks. The first block is the core test administrator, which manages the "Graphical User Interface", the "Test Administrator Database" and the spawning of all Distributed Test Administrators. The second block is a distributed test administrator, which is associated with a test generator and manages the interfaces to the test sites, the test generator and the core test administrator. The Core test administrator can spawn multiple distributed test administrator each associated with a different test generator and/or platform. What and where distributed test administrators are spawn is pre-configured in a validation system start-up file (viva.rc).

FIGURE 2. Test Administrator Architecture



As shown in Figure 3 on page 8, the distributed test administrator's relationship to a test generators is represented. The core test administrator will spawn off the distributed test administrator (DTA) and set up a communi-

cation interface between the two parts of the test administrator. The DTA will spawn off the test generator and test sites as pre-configured in a validation system start-up file (viva.rc). The DTA and test generator will always reside on the same physical machine.

As part of the spawning of the test generator a communication port is set-up, in which a request for a number of tests to be generated is issued. These tests and associated files are placed in a director stated be the DTA. When a test is created, the test generator notifies the DTA. At this time the test is read from the file system database and sent to a test site. The results of the test are sent back to the test generator via the DTA. The results are then examined by the test generator and reported to the DTA. If the test has no fault, it is removed from the file system database. If there is a fault in the test results, the appropriate files are removed from the file system database and sent to the test administrator's database to be used at a later time.

FIGURE 3. Test Generator/Distributed Test Administrator Relationship

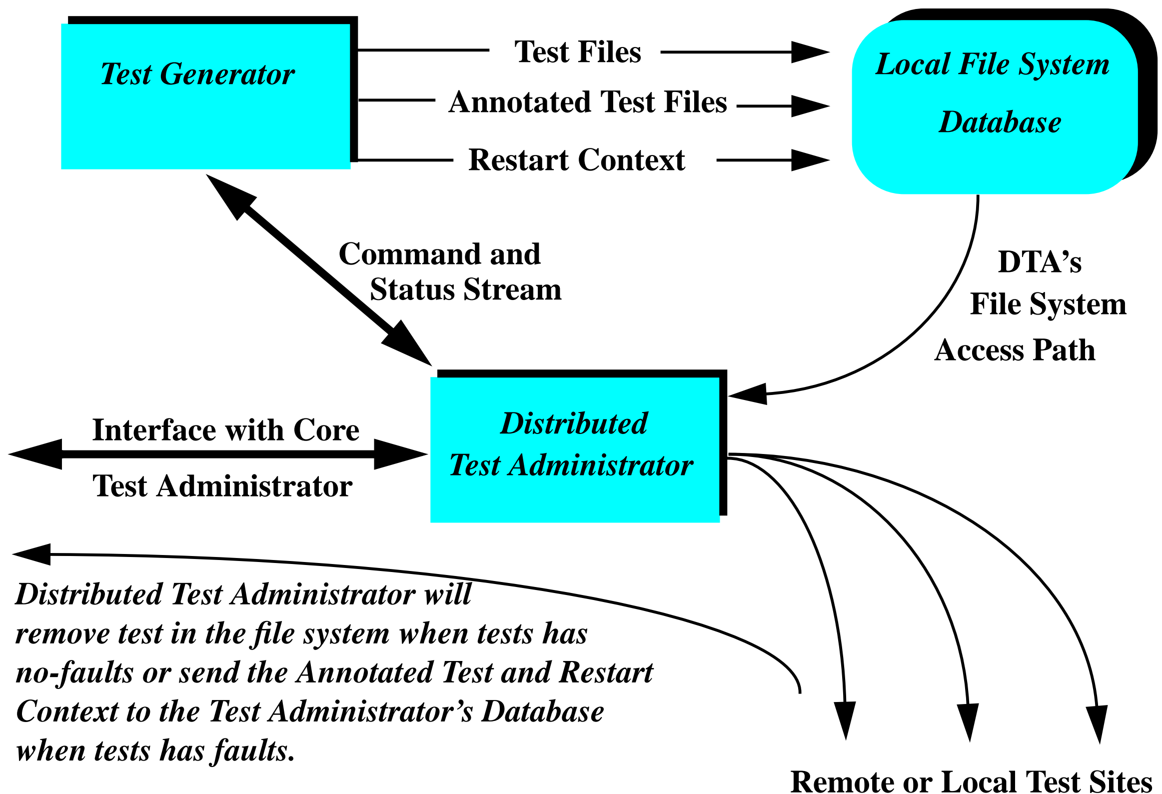
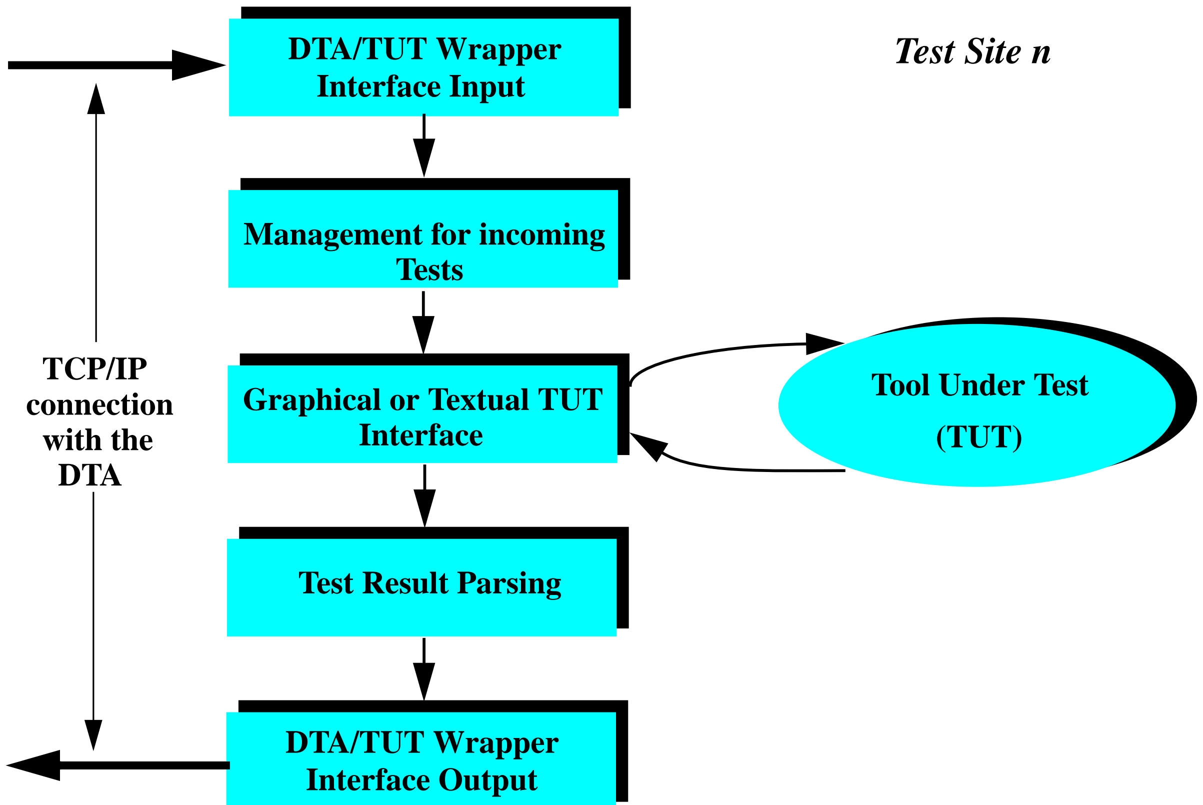


Figure 4 on page 9 illustrates the internal architecture of a test site. As part of the spawning of the test site a communication port is set-up between the test site and the distributed test administrator. This interface module will receive the test packets and set them up for use. The tool under test interface module will then select a test and run it on the tool. This module is unique to each tool. The tool must be controlled via an interactive interpreter controlled from within the application or via a graphical user interface requiring an "intercept" layer between the tool and user interface server by which the test site wrapper can introduce and observe communication between the tool and tool's user interface. Such an intercept layer involves compiled C++ code customized to the tool under test's 32-bit Windows or Motif graphical user interface implementation. It is up to the tool tester to develop this interface. The results from the tool will be parsed down to a string of keywords, phrases and characters that will allow the test generator to isolate a possible error. This string will then be sent back to the test generator via the distributed test administrator.

FIGURE 4. Test Site Architecture



Execution of the test sites wrapper is solely limited to platforms for which a Perl interpreter is available [1]. Interface to such tools under test is done by customizing one or more Perl scripts to the tool interface (test site wrapper), as mentioned earlier.

Due to high data volumes and a desire to interactively prune test generation in response to evolving test results, the preferred operating mode concurrently runs the test administrator(s), test generator(s) and wrapped tools under test (test site). In order to support off-line testing, a trivialized test site may simply capture every file sent to it or selective files, retaining such test cases in a file system or archival media (perhaps via an archiving mechanism such as tar or zip), and responding to the test administrator that the test was presumed to execute correctly. Replay of such validation suite tests may then be handled as with conventional tool validation processes. Without pruning by the trivialized test site, the validation system may readily produce test case volumes which substantially exceed the capacity of even large archival storage systems (tens to hundreds of gigabytes).

The test administrator(s) and test site(s) communicate via rsh operating system services (on start-up) and connected sockets once running. Rsh requires a running operating system service or daemon configured to permit service to the test administrator's computers and accounts[2]. Bidirectional connected sockets communicate a queue of test cases to the test site wrapper and return a queue of responses (in VIVA-specific form abstracted from a tool by the test-site wrapper) to the distributed administrator requesting test case execution. There is **no** assumption that the distributed test administrator and test sites share a common file system or share memory address spaces (an underlying TCP/IP implementation may often use such mechanisms to increase communication performance, especially if the test administrator and test sites are co-located on the same computer). A queue of tens or hundreds of test cases may separate test generation from failure responses back to the test generator.

The distributed test administrators and test generators communicate via system calls initiating a child processes (the test generator) with specific command line arguments. Subsequent communication is via files written (or read) in the local file system and control flow commands sent via redirection of the test generator's standard input and standard error streams. The architecture assumes that the test generator and the distributed test administrator are executing on the same computer, with closely related operating system permission configurations.

There is no assumption that both the core test administrator, distributed test administrator/test generator and test sites are operating on the same computer architecture or operating system. Test sites are *grouped* with a test generator and platform of test. This is not to say that the test generator and test site must be on a platform of like characteristics, but all the test sites associated with that test generator These parameters are delivered to the test administrator via the *viva.rc* start-up file or graphical user interface. For example, variations may include computer architecture variations (such as SPARC Version 8 and Version 9), operating system variations (such as SunOS 4.1.4 versus Solaris 2.5.1), storage configurations or the presence of alternative tool versions. The test

1. Note that the Perl interpreter is not developed or deliverable as part of this project and thus is subject to any license terms specified by the Perl provider. Perl is generally available without cost and is aggregated on the distribution media solely as a convenience to those receiving the distribution.
2. Note that the test administrator's account has no required relationship to NT's default configuration "administrator" account. Any account with required permissions is sufficient.

administrator does assume that a group selected for a particular test run is potentially subject to the same validation tests (test branches pruned by validation failures may eliminate some validation tests early in the verification process in order to reduce the cost of failed validations).

Execution of the test administrator and its graphical user interface requires either a 32-bit Windows or Motif graphical user interface and a x86/Windows NT, x86/Linux, SPARC/SunOS, SPARC/Solaris, HP PA/HP UX or PowerPC/AIX computer configuration[1]. If all failed tests are retained in annotated form, substantial persistent storage may be required on the core test administrator's machine in the form of disk space or on-line tape device(s). Note that the database utilized by the core test administrator is specific to this tool and does not require any external, general-purpose database management system.

Requirements for test generators depend on the team developing the specific generator. Execution of verification test generators developed by FTL Systems, Inc. requires a x86/Windows NT, x86/Linux, SPARC/SunOS, SPARC/Solaris, HP PA/HP UX or PowerPC/AIX computer configurations[2], whereas those generators developed by Clifton Labs, Inc. are executable on any platform for which a Java runtime environment is available, including x86/Windows NT, x86/Linux, SPARC/SunOS and SPARC/Solaris computer configurations[3].

Validation of a tool under test is generally a lengthy and thus expensive computing task. Extensive validation can expected to consume tens to thousands of computer / days. Completion is generally expected by a specific deadline. Over tens to thousands of computer / days and intentionally broad verification conditions, individual failure events within the validation system are ***probable***. Thus the validation system architecture must detect and transparently recover from individual failure events as far as feasible[4]. The system architecture should remain robust to the following failure events within the validation system:

- arbitrary failure of the tool under test,
- non-responsive or transient failures of the network,
- non-responsive failure of a test site computer,
- non-responsive failure of a tool under test,
- overly responsive failure of a tool under test (subject to operating system support),
- non-responsive failure of a test generator,

1. Note that platforms are being phase-in by demand; platforms without demand will be omitted
2. Note that test generator platforms are being phase- in guided by demand; platforms without demand may be omitted.
3. Note that the Java runtime system is not developed or deliverable as part of this project and thus is subject to any license terms specified by the Java vendor. Java implementations are generally available without cost and are aggregated on the distribution media when permitted by the applicable license agreements.
4. Any modification to the implementation can alter this fault-tolerance and thus is done completely at the risk of the party performing the modification.

- overly responsive failure of a test generator (subject to operating system support), non-responsive failure of the computer running the administrator provided the persistent file system is intact and consistent as of the last validation database checkpoint.

Some of the above internal failures may preclude completion of the specified validation activities (for example a network failure may preclude access to all computers specified in the SPARC Version 8 computer group). Under such failure conditions, the administrator should describe the failure to its user interface in sufficient detail that the average user can proceed with diagnosis, ideally allowing restart of the interrupted test sequence from the last database checkpoint.

The administrator functionality (including its graphical user interface) and administrator file system represent a potential single points of failure. Algorithmic flaws or programming flaws within the administrator can lead to apparent internal system failure manifest at the validation system user interface as incorrect results or non-responsive operation. In order to minimize such failures, the administrator interfaces and algorithmic functionality must absolutely minimize special cases impacting its control flow. If a failure does occur the testing can be pick up from the checkpoints stored in the "Validation Status Database". Failures in the administrator file system, as supplied by the operating system on the computer running the administrator(s), should be addressed external to VIVA via mirrored file systems, error-correcting file systems or periodic checkpointing to backup media.

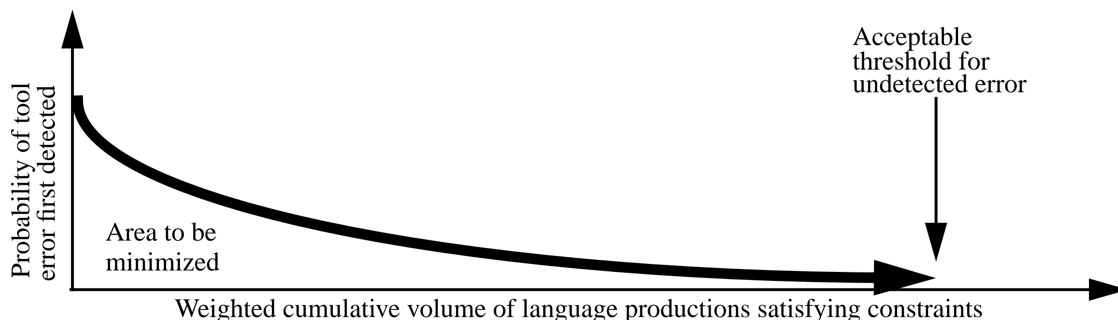
General Approach to Test Generation

The approach to test generation used by VIVA conceptually enumerates all language productions consistent with a set of grammatical, semantic, sequential and concurrent assertions as tests of correct VHDL ("correct test cases") and productions close to but incompatible with the assertions as tests of error handling ("error test cases"). Such generation of language productions is conceptually the inverse of compilation and execution, allowing *adaptation* of many existing compiler techniques.

Unfortunately, for VHDL and almost any other practically useful language, the set of all such language productions is infinite. As one of many examples, VHDL allows an infinite number of signal declarations within a block declaration. Thus the set of all language productions satisfying the specified assertions **cannot** be completely generated, captured on storage media or applied to a tool under test within any finite verification time interval or using a finite pool of computers.

Thus our objective in language production based on constraints is to minimize the probability distribution function (PDF) area for first detection of an arbitrary percentage of all errors (validation confidence level) in the tool under test. This objective is shown graphically in Figure 5 on page 13. VIVA test generators achieve this objective via heuristic mechanisms guiding and pruning language productions. The rest of this section will examine the design objectives which these heuristic mechanisms seek to optimize and some of the general heuristics employed by all of the generators. The following four sections will go into domain-specific constraint specification approaches and heuristics in greater detail. Subsequent chapters will detail the techniques utilized by VIVA.

FIGURE 5. Qualitative form of function production heuristics seek to minimize.



The cost of applying a language production to a tool under test has both a fixed cost component (tool start-up) and a variable cost component which increases with increasing length of production applied (although generally not in any simple relation). Thus the cumulative volume of language productions (X Axis of Figure 5 on page 13) must be weighted to account for tools start-up costs associated with each discrete language production (test case), resulting in a pressure to increase the length of each test case. Conversely, the presence of an error early in a production may mask detection of a subsequent error (for example the first error may cause the tool under test to terminate execution immediately, and thus the second error will never be reached).

From both inspection of Figure 5 on page 13, in which the curve asymptotically approaches 0%, and the inevitable flaws in human-generated language constraint specifications, the reader should conclude that **absolute** confidence in the validation of a tool under test is not practically achievable. The number of undetected flaws in human-generated language specifications may be reduced through terse specifications (assuming the probability of error increases almost linearly with the number of lines of specification text) and completely independent development of specifications for the same VHDL language functionality (assuming errors in specification entry are essentially uncorrelated). Both assumptions are believed sufficiently correct to warrant use of terse constraint specifications and two, redundant, independently developed constraint specifications.

Lexical and syntactic grammars provide the structural framework onto which the subsequent specifications build. Constraint values are inherited up and down the grammar production trees (as summarized in Table 1 on page 14). At each production, constraints must be satisfied for production of a correct test case or may be intentionally violated to generate an error test case. Pre-execution semantic constraints are introduced at the level of expressions (guided by inter-statement relationships, as in an assignment, and visibility ambiguity constraints). Sequential constraints are introduced at the level of sequential declarations and statements (such tests do not rely on inter-process communication). Concurrent constraints are introduced at the level of concurrent declarations and blocks (potentially inherited from one block down to the next through components, entities, architectures and configurations).

TABLE 1. Constraint values utilized at different levels in the lexical and syntactic production hierarchy.

	Concurrent Statements	Sequential Statements	Expressions
Lexical, Syntactic and Pre-execution Semantic Test Generation			***VIVA_Kind VIVA_Type, VIVA_Ambiguity, VIVA_Static***
Sequential Functionality Test Generation		***VIVA_ProcessState VIVA_ControlFlow VIVA_Value***	***VIVA_Kind VIVA_Type, VIVA_Ambiguity, VIVA_Static***
Concurrent Functionality Test Generation	***VIVA_Events***	***VIVA_ProcessState VIVA_ControlFlow VIVA_Value***	***VIVA_Kind VIVA_Type, VIVA_Ambiguity, VIVA_Static***

A class definition represents the state of an constraint value, methods used to derive other constraints within the grammar and methods used to locally evaluate the constraint to determine locally allowable lexical and syntactic productions. Following sections overview pre-execution semantics, sequential constraints and concurrent constraints. Subsequent chapters will detail the information contained in each class and the class's method signatures.

Test Generators for Lexical, Syntactic and Pre-Execution Semantics

Test generators for lexical, syntactic and pre-execution semantics provide both the structural framework and test characterizations on which subsequent test generators are based. For example, the liveliness test of an assertion statement may assume that previous operation of a lexical/syntactic/pre-execution semantics test generator has explored the space of boolean expressions governing the activity/inactivity of the assert statement. This allows the liveliness test generator to concentrate on I/O characteristics of the assert statement using relatively trivial boolean expressions. The sequential and concurrent test generators subsequently operate by inserting additional assertions into the syntactic grammar describing VHDL.

Lexical and syntactic tests are generated by choosing particular alternative productions (strategy) during top-down traversal of the grammar (from a start symbol). Different choices result in distinct tests. Choices must be made from character set classes (denoted by characters enclosed between '[' and ']'), alternatives (denoted by '|'), and iterative operators (See "Test Generator Language Specifications" on page 35.)

Pre-execution semantic tests are specified by associating constraints with expressions and their child productions (such as names). Constraints on expressions generally take the form of a kind constraints (for example the expression is a signal), (sub)type constraints (for example the expression's type is bit), ambiguity constraints (for example the name may have multiple overloaded meanings) and a statisticity constraints (for example the expression is locally static).

As constraints are passed up and down the tree, the constraints are generally transformed. For example, type constraints on an expression production called within a delay clause may initially constrain the delay to type time with a subtype ranging from 0 to time'high. Further generation of the expression production will eventually encounter several optional dyadic operations. If the optional dyadic operation is selected by a particular language production, there will be generally be different constraints on the left and right operands. Furthermore the type constraint returned from the left operand grammar will generally further constrain the right operand production.

In order to expedite test generation, all constraints are incrementally and recursively applied to control the set of correct and incorrect language productions at each step in the production process in order to achieve the desired test case objectives. For example, during lexical production of a locally correct VHDL test case, an assertion that a based literal lexeme has a subtype between 0 and 15 will permit generation of any character sequence compatible with the base subtype assertion; then return the chosen value for use during production of the integer part. When the production generation arrives at the integer part, production will be constrained locally by the conjunction of the previously chosen base and the resulting subtype domain. It would be much less efficient to generate complete based literals (in this example), then test the literal for compatibility with an assertions.

As noted earlier, a validation system should ideally have more than one implementation of the lexical, syntactic and pre-execution semantics test generator installed (to reduce the number of undetected flaws in the validation process). In order to find errors earlier, it is desirable that such test generators apply different heuristics and/or utilize functionally distinct language specifications (perhaps inverting the order of productions in one).

Test Generators for Liveliness

Liveliness test generators explore the ability of a tool under test to interact with the test wrapper and operating system environment via textio (VHDL “read” and “write” functionality), assertions, reports, storage allocation and storage deallocation. Results from applying liveliness generators to a tool under tests help to prune operation of subsequent test generators. In the most extreme case, few tests can be effectively run on a tool unable to respond to the wrapper via proper execution of textio, assertion statements or report statements. If failures are limited to formatting, writing, reading or scanning specific data types or execution of specific statements, subsequent tests can rely on I/O mechanisms demonstrated to work during the liveliness phase.

FTL Systems’ liveliness test generators use a Perl script to customize and arrange parameterized VHDL fragments related to textio or dynamic-memory allocation into a stream of correct or error test cases. VHDL fragments may either be embodied directly in the Perl script, read from a file or supplied via optional parameters on the command line (as motivated by the following paragraph).

Many alternative implementations of liveliness tests are possible, including reference to a manually prepared archive of test cases (precludes several test forms described below) or automated generation from a textio or allocation-related specification grammar. There is nothing in the VIVA architecture requiring or predisposed to use of Perl for liveliness tests.

Subsequent test generators which validate the sequential or concurrent functionality of a VHDL-related package, such as a multi-valued logic package, with internally defined textio subprograms and implicit allocator/deallocator functionality for newly defined types, may recursively utilize liveliness test generators. Such a recursive test generation process may effectively utilize optional parameters on the test generator command line to specify the new types and test sequence numbers to be utilized by the child test generation process.

Liveliness tests are also the first of the four test generators described here in which the test case includes both VHDL code intended directly for the tool under test and an expected results specification. The expected results specification appears at the beginning of the annotated test file (set off as comments preceded with three leading ‘-’ symbols) in the form of assertions which must evaluate to true in order for the test result to be judged correct. The expected test results are generally **not** a text string which must be matched with the tool under test’s output (“Interfaces Between Test Administrator and Test Site Wrapper” on page 77).

Liveliness validation tests are particularly sensitive to details of the computer configuration running the test-site. Parameters such as physical memory, virtual memory, availability of a writable file system, the operating system version, the processor architecture version and even other applications running concurrently may impact the success of a liveliness validation test. Thus strong consideration should be given to running liveliness validation with a wider range of computer configuration groups (via the test administrator) than would be recommended for other validation test generators.

Test Generators for Sequential Functionality

Test generators for sequential functionality implement testing of sequential-statement control flow, sequential statement operation and evaluation of expression values during tool execution (for example, simulation). Such tests do not involve any form of communication between VHDL processes. Test cases may contain more than one VHDL process if possible error masking is taken into account by the overall test strategy.

There are fundamentally two kinds of sequential functionality test generators; those relating to the correct implementation of intrinsic VHDL language constructs (such as the addition of two integers defined in package standard) and those relating to the correct implementation of packages, package bodies, entities and architectures which are specified using VHDL (such as the VITAL timing library). The first kind of tests will be referred to as *intrinsic* tests, the second *extrinsic*.

For intrinsic testing, validation begins with assignment expressions testing the numerical correctness of simple expressions (using the results of previous liveliness testing to guide use of assert statements, report statements and/or textio). Sequences of more complex sequential statements, control flow and subprogram calls are added (in order). Expression formation uses symbolic evaluation, knowledge of common computer architecture data types and an infinite precision mathematical package in order to select useful test cases and define expected results.

For extrinsic testing, the functionality to be tested must be fully specified within a prototype implementation (generally derived directly or manually from the desired VHDL-related standard). VIVA then compares the functionality of the tool under test to the functionality which would have been predicted by use of the prototype implementation. Obviously the prototype implementation must be completely defined (all deferred constants defined, all subprogram bodies defined, and all components bound to a defined entity and architecture pair). Extrinsic tests also presume prior validation of sequential and concurrent intrinsic VHDL functionality.

Knowledge of VHDL control flow operators, process state and expression value is embodied in ***VIVA_ControlFlow***, ***VIVA_ProcessState*** and ***VIVA_Value*** classes and assertion values of these classes (See “Sequential Intrinsic Function Test Generation” on page 57.). These assertions are propagated top-down and bottom-up through productions of the lexical and syntactic grammar in order to control choice of alternative sequential statement and (generalized) expression productions, augmenting assertion values already implemented for test generators described previously.

Verification of intrinsic, sequential functionality within VHDL processes in turn provides a foundation on which we may build validation of concurrent intrinsic and concurrent extrinsic functionality. Test generators emitting such tests are described in the next section.

Test Generators for Concurrent Functionality

Concurrent test generators produce test cases verifying that VHDL processes inter-act in accordance with language-architected communication mechanisms. VHDL provides for both signals and shared variables as inter-process communication mechanisms.

Just as with sequential functionality test generators, concurrent test generators are divided into intrinsic and extrinsic versions. Intrinsic versions explore a tool's ability to implement language mechanisms defined by the base VHDL language whereas extrinsic versions explore implementation of communication mechanisms which may be embodied in a proprietary (perhaps optimized) implementation of a VHDL-related standard (such as Vital).

Top-down and bottom-up assertion propagation also provides the mechanism for concurrent functionality verification. Assertions are made concerning side-effects initiated by one or more processes and visible to other processes. The side effects may be assignments of the same value to a signal ('active side-effect), change in signal value, resolution of signal value, assignments of the same value to a shared variable or assignments of a new value to a shared variable. The side-effect event(s) may be visible to another process by virtue of triggering execution (process sensitivity list), reference to a value (conceptually on the right hand side of an expression) or suspension (with some monitor approaches).

Just as kind, type, ambiguity and statisticity are propagated up and down expression production trees for pre-execution semantic verification, assertion values of type ***VIVA_Events*** propagate up and down VHDL grammar rules to implement concurrent functionality. These values represent a partial ordering of (generalized) events and value relationships visible across process boundaries. They are generalized in that they represent events in the history of signals as well as shared variables (and eventually quantities if extended to VHDL-AMS).

Generation of concurrent functionality tests begins at a concurrent block level, where assertion functionality annotating the language grammar expresses the event interactions permitted for correct VHDL, VHDL test cases with errors that must be detected and VHDL test cases with errors that should never be written but need to be detected by a tool implementation. Analogous to the choice of alternative grammar productions at the syntactic level, concurrent interaction strategies may be chosen at the concurrent block level and propagated downward into concurrent statements, sequentially statements and expressions as derived assertions. Event partial ordering assertions are passed by value into and out of grammatical productions using exactly the same mechanism as is used for kinds, types and other assertions.

The internal representation of ***VIVA_Events***, embodied as a class, resembles a generalization of VHDL's projected waveforms (See "Sequential Intrinsic Function Test Generation" on page 57.). The number of waveform elements, their temporal relationship (ordering and time from one element to the next) and the objects they side-effect (such as shared variables) are more general than those permitted for a VHDL projected waveform.

General Approach to Test Administration

Test administration provides an organization for coupling multiple test generators, multiple test sites, the database of test results and the (graphical) user interface so as to realize a fault-tolerant VHDL tool verification system. For configurations of approximately ten or fewer computers, the test administrator is designed to run as a single operating system process, potentially time-sharing a computer with test generation or test-sites.

A (graphical) user interface needs to beable to alter or define the configuration of the test generators, configuration test-site computer groups, and report of current test status. Each test failure is graphically representable as a stack of hypertext links into the appropriate language reference manual, beginning with the immediate point of failure. At each point in the stack there may be one or more relevant references explaining where the test case was formed from in terms of standardization text.[1] A suitable[2] HTML browser is required for such display, but not included in VIVA distributions.

The test administrator is responsible for keeping track of on-line test generators (those test generators potentially capable of producing test files in response to a command), test cases which have been generated but not successfully completed on all computer groups, on-line test sites (those test sites potentially capable of accepting and or responding to a test file), subsystem failure detection/recovery, summary report generation and graphical user interface activity (keeping the user updated on validation progress).

In order to track and report tests which have been generated and yet not completed successfully by a test-site from all computer groups, the test administrator requires a persistent database associated with a specific tool verification process. In order to provide for portability among various computers and reduce licensing restrictions, the test administrator uses a simplified persistent database specifically designed to retain the status of tests indicating a flaw in the tool under test (VHDL with an undetected error and correct VHDL triggering a tool error).

Since test administration is potentially a single point of failure, it is essential that the test administrator be logically as simple as possible by localizing special-case program logic within the test generators and test site wrappers whenever possible. Nothing internal to the test administrator should be hard-wired for either VHDL or a specific tool under test.

The test administrator database periodically forces a consistent update of test status and the latest level of pending test_recovery files for each test generator to one or more operating system files. For performance reasons, the test administrator retains a cached copy (in memory) of test status, computer resource configurations and the latest test_recovery files for each test generator. A pair of sequential-access database files for each computer group configuration running tool verification are alternatively updated so that one complete database (perhaps some-

1. Licensing restrictions imposed by IEEE limit redistribution of on-line VHDL Language Reference Manuals.
2. Initially versions assume a Netscape Version 3.0 HTML browser. Unfortunately no standard yet governs portable, external positioning of the reference point within a displayed HTML document, hence please use Netscape for now.

what dated) will always be closed and complete on rotating media (swing-buffer approach), increasing the probability of rapid (perhaps 1 to 10 minute) recovery even from catastrophic and unpredicted computer failures.

Organization of Subsequent Chapters

This chapter provided a summary of the VIVA architecture, algorithmic approaches and interfaces. For details required for integration, please refer to one of the following chapters providing much greater detail:

- Interfaces common to all test generators (Chapter 2),
- Test generators language definition for lexical, syntactic and pre-execution semantics (Chapter 3),
- Test generator's usage of language definition for lexical, syntactic and pre-execution semantics (Chapter 4),
- VIVA installation (Chapter 5),
- annotated bibliography describing related work (Chapter 6).

This document is distributed in both Postscript[1] and PDF[2] formats. Postscript copies of the document print as 94 pages. PDF versions of the document provide extensive hypertext links (denoted by highlighted text). Use of an on-line PDF version of the documentation is therefore encouraged.

1. Unfortunately, a flaw in development tooling currently precludes display of the Postscript version using GNU Ghostview. A fix is expected before production release which enables use of Ghostview.
2. Note that a PDF reader is not developed or deliverable as part of this project and thus is subject to any license terms specified by the PDF reader provider. A PDF reader is generally available without cost and is aggregated on the distribution media solely as a convenience to those receiving the distribution.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

CHAPTER 2

Interface Common to All Test Generators

This chapter defines the interface requirements which any test generator must comply with in order to inter-operate with the test administrator. Sections of the chapter describe the interface architecture, common test generator command line options, interactive control commands, file output requirements, status responses from the test generator, fault tolerance requirements and software engineering requirements.

Test generators may take many forms including both the four versions explicitly specified in this document and a wide variety of others not explicitly provided with VIVA. For example, a test generator could be developed which supplied its test sequence from a pre-stored archive (for compatibility with legacy or special-purpose test cases). Test generators could readily be developed or adapted from prior work in order to measure a tool's response to capacity stress, its performance, a tool's ability to gracefully handle random permutations of VHDL or other text or even exercise seldom-used pathways within the tool.

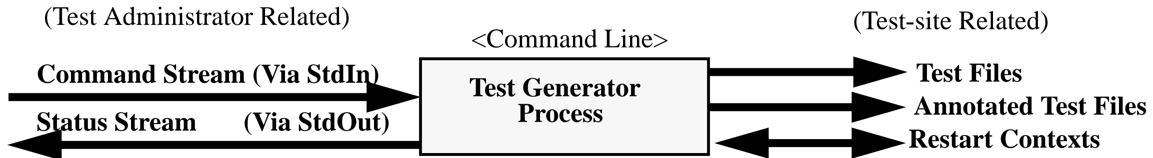
Test generators may be written in a wide variety of languages, including C, C++, Perl, Python, Java, Basic and even database languages such as SQL with appropriate encapsulation. Generally, the language chosen for the test generator need only be capable of yielding an executable which can be exec'd via an operating system call as an independent operating system process (perhaps in conjunction with an interpreter as in the case with Perl), which is capable of reading interactively from a standard output stream, and which is capable of reading and writing a sequence of files to the host operating system.

The interface between the test generator and its environment was specifically designed to be simple, to be readily adaptable to existing validation techniques, and to be readily exercised independently from the rest of the VIVA tool environment. This simplicity, adaptability and portability does come at modest performance cost.

Interface Architecture

Interfaces to and from the test generator are shown in Figure 6 on page 24. A functional test generator integrated into VIVA must map into this interface architecture. Implementation within the test generator process is entirely governed by the requirements for a stand-alone executable process on a given computer. For further information on such processes, please refer to system-level programming documents for the computer in question.

FIGURE 6. Interfaces to and from test generator process



A set of command line arguments provide initial, mandatory configuration parameters for the test generator (See “Command Line Control Flags” on page 26.). Additional configuration parameters may be supplied using optional, test-generator specific command line arguments, environment variables, and test-generator specific start-up files.

The test generator process corresponds directly and interactively with the distributed test administrator using standard input (See “Command Stream Control Flags” on page 28.) and standard output (See “Status Stream Control Flags” on page 30.). The command stream regulates the production of test (flow control) and test results so that the test generator can dynamically prune the test generation tree. The status stream from the test generator supplies test statistics for the test administrator’s database (See “Test Administrator Database Structure” on page 75.), which in turn supplies a report to VIVA’s graphical user interface.

The test generator emits three different, test-site related files in the local file system for each test generated. The three files are contained in different directories and related by a common file name to the left of the ‘.’ character. The ‘Annotated Test File’ must be emitted before the status stream’s test_done command is issued to the distributed test administrator[1]. When a test is successful, the files are deleted by the test administrator. When a test has a fault, the test administrator saves the annotated version of the test and the restart context file. Distribution (and redistribution[2]) of test files to test-sites is the responsibility of the distributed test administrator. Further details on the files emitted by the test generator may be found later in this section (See “File Output Requirements” on page 32.).

1. The annotated file may not be emitted at the same time as the test file, so that it may add the results of the run reported back to the test generator from the tool under test.
2. Redistribution is required to handle test-site and network failures.

When a test generator completes, a return value 0 (zero) indicates that all was successful. This does not indicate that there was no faulted test, just that the test generator performed as expected. If the test generation process was not fully successful, the test generator should return a non-zero error code corresponding to one of the error values specified in the computer's <errno.h> header supplied with the computer's operating system.

Command Line Control Flags

This section specifies both mandatory and optional user command line interface flags which all test generators must implement. Test generator specific command line arguments may be supported and can be initiated within VIVA from either the graphical user interface or test generator specific start-up file.

Command line flags resemble the following example:
-test_directory '/usr/viva/test/' -testa_directory '/usr/viva/testa/'
or
-test_directory 'd:\viva\test\' -testa_directory 'd:\viva\testa\'

where any file names provided as arguments will be quoted in order to avoid possible expansion by a shell or other command interpreter (addresses test generators implemented using a shell script). Directory names will contain the trailing '/' or '\' (depending on the operating system) so that generated test names can be concatenated to the directory argument with out knowledge of operating system the generator is running on.

In general, a command is preceded by a single '-' and followed by an argument which does not begin with a '-'. The argument and the subsequent command are delimited by one or more white-space character(s). The white space characters are a blank or tab. White-space characters are not permitted within a command or argument. An embedded semicolon should not appear within a single command line for shell compatibility.

The required user commands are:

-test_directory <directory_specifier>
denotes the directory into which test files should be placed.

-testa_directory <directory_specifier>
denotes the directory into which annotated test files should be placed.

-restart_directory <directory_specifier>
denotes the directory into which restart contexts should be placed.

The optional user commands are:

-language <label_of_predefined_language_option>

denotes the language which should be generated if multiple languages are available. Current options include VHDL93, VHDLAMS, STDLOGIC and VITAL. If not specified, a default language will be selected by the test generator.

-generate <integer_literal> <"bad" or "good">

denotes how many and what kind of test files should be generated on start-up. This option is used for a batch run, once the number of test are created the test generator exits. If there is no '-generate' command specified in the 'Command Line', then after the test generator process is started that process will stall until a generate command is received on the 'Command Stream'. A bad test will be generated with a known error to assure an analyzer under test is checking thoroughly.

-conformance

this command changes the waypruning is done. If this flag is set,the test will be generated as if no faults have occurred. If the fault threshold for a production is reached, then the test which includes that production is discarded and not registered with the test administrator.This will allow for the predictable recreation of tests, independent of a tool under test failure. If this flag is not set, reaching the fault threshold for a production can alter the direction the test generator goes in generating a test.

-restart_checkpoint_interval <integer_literal>

denotes interval of restart context generated by the test generator. If not specified '1' should be default. ***Warning****: This flag should be used sparingly,because it limits the ability to restart test generation at a particular test*

-restart_from <file_name_in_restart_directory>
denotes the file name within the restart directory at which test generation should resume. The test generator should verify that the restart file was one it had initially created or is upward compatible with the current version of the test generator *(See "Software Engineering Requirements" on page 38.).*

If more than one instance of a particular command flag appears on the command line, the test generator should ignore all but the last such instance of the command. If a specified directory or file does not exist or is not writable for a non-ignored command, the test generator should terminate with a suitable error number. Requests to generate less than zero test files should be interpreted as a request to generate 0 test files. Use of a non-integer argument should cause termination with a suitable error number. Optional command line arguments which are not recognized by the test generator should be ignored (for upward compatibility). A termination due to an invalid command line should be explained by a message on the status stream to the test administrator using the '-ui_message' command prior to the test generator's termination. The message should be sufficient to allow the user to isolate and repair the failure. For FTL-developed test generators, please use internationalized message format.

Command Stream Control Flags

The control stream interactively communicates from the distributed test administrator to the test generator using the test generator's standard input stream. Some messages in the control stream are common to all test generators, however nothing precludes a test generator from implementing domain-specific commands.

The format of control messages closely resembles that of the command line. Each command begins with a '-' and may be followed by zero or more arguments. The blank and/or tab characters separate commands and arguments. The command stream is terminated with a ';' character. The command stream should be read and executed by the test generator when a ';' is seen on the command stream if the generator is idle. SIGINT will be used to interrupt the test generator if a command has been sent and the generator is in the process of generating a test.

The following control flags (commands and arguments) will be implemented by all test generators:

-restart_from <file_name_in_restart_directory>
denotes the file name within the restart directory at which test generation should resume. The test generator should verify that the restart file was one it had initially created or is upward compatible with the current version of the test generator. This will interrupt the generation of test if the test generator is currently generating test.
(See "Software Engineering Requirements" on page 38.).

-generate <integer_literal><"bad" / "good">

denotes how many and what kind of test files were generated. If currently generating tests and a second generate command is received, standard error will be consulted by the test generator at the conclusion of current generate command before starting the newly issued generate command to assure that there is no problems. The number of test from the newly issued generate command will be added to the current number of test to be generated. If the type of tests are different for the current tests being generate, the test generator must keep track of this. Bad tests will be generated with a known error to assure an analyzer under test is checking thoroughly.

-test_results <test name> <string>

denotes the results of the test stated in the test name field from the tool under test. These results. will be returned as a string. The string is made up of messages separated by a '\n' (newline) character. A message will be in the format as follows:

<error serverity> :: <line number> :: <character offset> :: <string of key words>

The error serverity will be listed as one of the following words:

(FAILURE, ERROR, WARNING, NOTE)

If the tool under test does not supply character offset a '-1' will be indicated in this field. The first message in the string will be the number of messages in the string excluding the first.

-succeeded <quoted expression or subprogram signature> <test_name>

this command comes in response to an earlier query command on the status stream and denotes a liveliness capability which tested successfully.

-failed <quoted expression or subprogram signature> <test_name>

this command comes in response to an earlier query, denotes a liveliness capability which tested unsuccessfully.

-statistics

denotes that the running status of the test generator to be returned to the test administrator. *(See “Test Administrator Database Structure” on page 75.).*

-exit

directs the test generator to terminate testing and exits at once. The same error handling criteria noted for the command line apply to command stream (See “Command Line Control Flags” on page 26.).

Status Stream Control Flags

The status stream interactively communicates from a test generator to the distributed test administrator using the test generator's standard output stream. The distributed test administrator must handle the status flags (commands and arguments) noted below. All other commands are ignored (for upward compatibility).

The format of status flags closely resembles that of command line flags and command stream flags. Each status message begins with a '-' and may be followed by one or more arguments. The blank and/or tab characters separate commands and arguments. Since the status stream is read interactively, the presence of a ';' character in the status stream directs the test administrator to handle any prior flags. Since the status stream is the test generators standard output, the test generator *MUST* flush STDOUT buffer after a command is sent.

The following status flags (commands and arguments) will be implemented by the distributed test administrator:

-ui_message <quoted string>

denotes a message to be sent to the test administrator's graphical user interface and summary log.

-ui_statistics <quoted string>

denotes statistical data to be sent to the core test administrator's graphical user interface, if queried for, and test administrator's database.
(See "Test Administrator Database Structure" on page 75.)

-register <quoted expression or subprogram signature> <test name> <group>

this command is used to register a generated test with the distributed test administrator. If the test is a liveliness test it denotes the expression and test name associated with a particular liveliness functionality (for tracking success or failure when this function is queried at a later time.) When the test is not a liveliness test, the expression will be a NULL string. The test name and group will be specified. The group if NULL indicates a single generated test file in to this test. If there are more than one generated test files to this test, then there will be a identical group number for all files associated to the test. These tests will sent to the same test site and will be run in the same order that the test generator gives them to the distributed test generator.

test_done <test name> <"fault" /"no_fault">

this command is sent to the distributed test generator at the end of a test loop. The parameter 'fault', is sent when agood test is generated and had a failure, a bad test is generated and did not have failure, or a bad test is generated and the failure is not what was expected. The parameter 'no_fault' is sent when the results are as expected.

-query <quoted expression or subprogram signature>

inquires concerning the success or failure of the particular quoted expression or subprogram which denotes a liveliness capability. Test administrator responds with a succeeded or failed command stream.

-returning <integer error number>

denotes a programmed return of the test generator and an error number from errno.h denoting the reason for the abort. A zero return code denotes successful completion of all programmed tests.This does not indicate that there was no faulted test, just that the test generator performed as expected.

Handling criteria noted for the command line applies to status streams (See "Command Line Control Flags" on page 26.).

File Output Requirements

Test generators emit a sequentially numbered sequence of tests. For each test there are three files, emitted into three distinct directories denoted on the test generator command line. Each test generator is given a unique directory for each of the three kinds of test files. The exceptions to this are if the user chooses to use the restart_checkpoint commands upon calling the test generator or if the test being generated is part of a group. In the case of the '-restart_checkpoint_interval' the restart context will only be generated by the test generator with a test generated at the stated interval. This flag must be used with caution since the ability to restart at a particular test is forfeited. In the case of the 'restart_from' the directories that will be used for test and testa must be compatible with the test generator. See "Software Engineering Requirements" on page 38. In the case when the test is part of a group, only one restart file will be generated for the group of tests. This restart file will restart the generation of tests with the first of the group, since tests within a group are dependent upon one another, restarting elsewhere in the group should cause a failure.

Each file name is constructed from a leading lower case 't' followed by the hexadecimal number of the test[1], a '.' and a three-character file extension differentiating test files, annotated test files and test recovery files. No leading zeros are permitted within the hexadecimal test number.

The test file is the VHDL source intended for the tool under test. If the file generated is not part of a group then it should be self-contained by working with the assumption that the tool under test is starting from its initial state. If the file generated is part of a group then it must be noted so in its restart file, (See "Software Engineering Requirements" on page 38.) It must also be registered with the test administrator as one test of a group. Registration of the tests will be in the same order the tests need to be analyzed for proper results.(See "Command Stream Control Flags" on page 28.) A test's result will remain until the last test of the group is analyzed. Test files in general, will have the extension ".tst". For example, a test file might be named "t1234AC.tst".

The annotated test file augments the test file with assertions concerning the expected response to the test and incremental language specification items used to create a derivation path back to the reference VHDL standards. Test assertions are prefaced on a lines beginning with "---" followed by a character other than '-'. Derivation lines are prefaced with "----" followed by a character other than '-'. Annotated test files have the extension ".ats". For example, an annotated test file might be named "t1234AC.ats".

The test recovery file provides enough context that the test generator can restart test generation with the associated test. Test recovery files are completely defined by the test generator, however the test generator should recognize test recovery files it generated. Test recovery files are handled as binary (all eight bits per byte are significant). Test recovery files have the extension ".rec". For example, a test recovery file might be named "t1234AC.rec".

1. Hexadecimal digits 'a' through 'f' in the file name are to be lower-case.

A test generator must be prepared to have up to 1024 outstanding tests. Outstanding tests are those which have been generated yet whose test result are unknown. Tests which have no faults will be deleted by the test administrator from all three directories. Tests which have a fault will be managed and stored by the test administrator.

Fault Tolerance Requirements

System architecture fault tolerance requirements dictate that a test generator may fail at any time. The test administrator is responsible for detecting and transparently recovering from such failure whenever possible. This fault tolerance capability requires that the test generator meet several requirements.

The test generator must be re-startable when supplied with a compatible restart context file. The test generator is solely responsible for insuring that it is compatible with the restart context file and that the restart context file is complete with no detectable corruption. The restart context file will have a unique header that can determine compatibility between the restart context file and test generator. (See “Software Engineering Requirements” on page 38.) If the test generator detects a non-compatible or corrupted restart context file it should fail, reporting the error to the user interface by way of the -ui_message and to the test generator by way of the -returning commands on the status stream.

When a test generator initially fails, the test administrator attempts a restart of the test generator from the last known restart context file. If the test generator fails again on what appears to be the test number, the test administrator assumes a failure in the test generator, marks the test generator as off-line, and sends a message to the user interface. If the test generator rejects a restart_context file, the test administrator reverts to a prior restart_context file if available. If this second file fails, the administrator attempts a restart of the test generator from the beginning of its sequence (clean start). On restart, only the restart context is supplied; any test files or annotated test files associated with the restart have been deleted.

If the test generator exits (for whatever reason) without allowing for the completion of the test being checked, the test generator will be restarted with the restart context file whose test was last checked. The test administrator will clean up the directories prior to restart. If this happens again a message is sent to the user interface.

If a test site fails while running a test, the test will be retried by the test administrator. If it fails again the test administrator will send a ‘failure’ severity level to the test generator for this test. The test generator will return the on the ‘-test_done’ command <fault> so that the file will be stored as a faulted test.

For performance reasons, *please* keep the size of restart files small whenever possible. While a restart file may be somewhat trivially constructed by dumping and then undumping the test generator’s address space (as with Emacs and TeX), this approach generally results in a prohibitively high performance penalty. Generally the restart context should capture near-minimum information needed to restart test generation, perhaps several thousand bytes of information at most.

The test generator is responsible for any child processes it initiates. The test generator will assure that the operating system terminates these processes along with the test generator (any time the test generator terminates).

Any test generator found in non-compliance with the preceding fault tolerance requirements during integration quality assurance must be clearly denoted at the user interface as ***Experimental, use at your own risk***. Formal

distribution of such test generators with VIVA should be isolated in a contributed subdirectory so as not to be confused with test generators which have passed system test.

Test Generator Message Format

For FTL Systems developers, the internationalized message format will be used by the test generators. Other developers are encouraged to use this format as well.

This format is simple. When a faulted test is discovered, the Key words, Characters or Phrases (*i.e. PACKAGE, ENTITY, ';'*) which are involved in the fault and are part of the language standard under test, will be passed to an error method with the message index selection. The error method contains an array of all error messages that can be detected by the test generator. The message's format allows for a Key word, Character, etc. insertion. Not all messages require insertions. This will allow for the messages to be translated appropriately to the language of choice without altering the generator.

CHAPTER 3

Test Generator Language Specifications

Introduction to Lexical and Syntactic Specifications

This chapter specifies the format used to configure lexical, syntactic and executable semantic test generators for a specific language under test. The resulting set of test generators may be customized and grouped in many different ways to implement specific kinds of lexical, syntactic, functional or liveliness tests.

Test generators may be created from the specifications outlined in this chapter using compiled, interpreted or other techniques as long as the result is a test generator capable of complying with the test administrator interface (See “Interface Common to All Test Generators” on page 23.). This portability is intended to increasing fidelity of the resulting validation by providing permutations of the front-end specifications and generators.

The chapter begins by describing FTL Systems’ integrated lexical and syntactic grammar specification characterizing the language implemented by a tool under test. Program fragments (in languages such as C++ or JAVA) may be inserted into the lexical and syntactic grammar specification both to embody semantic characteristics of the language under test (*constraints*) and to implement particular test generation objectives (*strategies*).

Program fragments are inserted into the lexical and syntactic specification text indirectly (by reference) in order to facilitate generation of variant strategies by an external agent. Each program fragment inserted by reference generally implements a particular semantic constraint or test generation strategy, modifying parameter values, local state and global state accordingly. Incorporation of program fragments by reference also enables use of the same lexical and syntactic grammar specification with program fragments written in a variety of languages, such as C, C++ and Java.

Globally-scope, persistent state may be declared and referenced by program fragments. This state may include information such as the set of accessible symbols and their meaning, global test generation objectives and other information shared among all language productions.

The lexical and syntactic grammar format specified in this chapter includes several mechanisms describing language production control flow; alternatives and iterators. In order to emit a test case, the generator must make specific choices for potentially recursive instances of each alternative or iterative production. For each choice, zero or more program fragments, encapsulated as a function, may be inserted to make the desired choice. If no such fragments are given, a suitable random choice generator is inserted automatically during processing of the specification into a test generator.

Each production rule in the lexical and syntactic specification includes both input and output parameters. Input parameters communicate contextual information into a specific instance of the production rule (terminal or non-terminal) directly from the call site within the grammar at which the production was called. Output parameters return information to the call site (terminal or non-terminal).

For simplicity, FTL System's use of the lexical and syntactic grammar associates a common parameter signature with all language productions (terminals and non-terminal). The later sections of this chapter describe the types and common parameter declarations use for all such interfaces using C++.

Subsequent chapters build on the lexical, syntactic and parameter passing foundation provided by this chapter in order to generate functional and liveliness tests using insertion of specific strategy programming language fragments. The mechanisms defined in this chapter translate subsequent strategies into specific, textual test cases which can be applied to the tool under test.

Lexical and Syntactic Specification Grammar

Lexemes

This subsection describes the set of lexemes which may appear in a VIVA front-end grammar specification file. Any illegal lexemes must be reported to the user at the earliest point feasible.

VIVA specification files consist of a sequence of graphical characters compliant with ISO _____ (16-bit unicode). Such graphical characters include upper case letters (A-), lower case letters (a-ÿ), digits (0-9), special characters, space characters and other special characters. Such special characters include quotation marks (“), number signs (#), ampersand (&), tick (‘), left parenthesis ((), right parenthesis ()), asterisk (*), plus signs (+), minus signs (-), periods (.), slashes (/), colons (:), semicolons (;), less thans (<), equalities (=), greater thans (>), left square brackets ([), right square brackets (]), underlines (_) and vertical lines (|). The space characters include spaces, non-breaking spaces, horizontal tabs, vertical tabs, carriage returns, line feeds, and form feeds. Other special characters include exclamation marks (!), left curly braces ({), right curly braces (}), division signs ( ) and other less frequently used characters.

3.0.0.1 Comments

Comments may be inserted between any two distinct, non-comment lexemes in the specification file using either C or C++ comment style. C comments begin with a “/*” and end with a “*/” sequence; such comments may include one or more embedded newline characters. C++ comments begin with a “//” character sequence and terminate at the first subsequent newline character. Comments may begin in any column (they are not restricted to begin in some specified initial column).

3.0.0.2 Keywords

Several different identifiers, listed below, are reserved as keywords. The keywords are case-sensitive, however in order to facilitate readability, use of a case-folded keyword, such as #START, is discouraged:

#include
#program
#start
#separator
#function
#language
#terminal
#enumeration
#nonterminal

3.0.0.3 Delimiters

Special characters include:
‘[‘ and ‘]’ generally denoting character sets,
‘{‘ and ‘}’ delineate program implementation source,
‘(‘ and ‘)’ delineate parameter lists and association lists,
‘\’ denotes a quoted character,
‘@’ is used at control flow function call sites,
‘;’ is used to terminate elements,
‘:’ is used to separate control flow function signatures and bodies,
‘|’ is used to separate generally equivalent alternatives,
single quotes ‘ delineate each character literal.

3.0.0.4 Character_Literal

Single characters enclosed within single quotes, such as ‘a’ are scanned as literally as single characters. Any character preceded by a ‘\’ acquires its literal meaning, as in \”, which is interpreted as the quote character. <<Change references to Unicode>>

3.0.0.5 Character Classes

Character classes denote zero or more character classes, from which exactly one character must be chosen during language production. If the leading character of a character set is a circumflex (^), the character set specifies all characters NOT explicitly denoted in the character set. The character in character element is a graphic or escaped (eg \]) character.

character_class::=‘**[**‘ character_element { character_element } ‘**]**’

character_element::=**character**
|**character ‘-’ character**

3.0.0.6 String Literals

String literals begin and end with (double) quote characters, as in the example “this is a quoted string”. Any embedded (double) quote must be escaped, as in “this example has an embedded quote \”. “

3.0.0.7 Integers

Integers include digits and “_”, however an integer must begin with a digit and is always interpreted in base 10.

3.0.0.8 Identifiers

Identifiers include lower-case characters, upper-case characters, digits and ‘_’, however an identifier may neither begin with a digit nor an “_” character.

Grammar

This section describes the complete grammar used to specify VIVA lexical and syntactic specifications. The specification format combines lexical and syntactic information, inter-mixed, within a single file. In the following grammar, lexemes introduced above are shown in bold. Comments can occur between any two lexemes, however they are not explicitly shown in the grammar (to improve clarity). Isolated character literals used in the grammar are quoted (quotes do not appear in the actual file). Productions in braces can occur zero or more times. Productions in square brackets can occur zero or once.

3.0.0.9 Specification Files

Most useful VIVA lexical and syntactic specifications consist of many specification elements, contained in one or more specification files. Each specification file consists of at least one specification element. Some specification elements define declarators which are subsequently referenced by other specification elements. Since the definition must precede use in VIVA specifications, the partial ordering of some specification elements within the hierarchy of file inclusions is significant.

specification_file::=specification_element { specification_element } <EOF>

Each specification element has a specific form, identified by the '**#**' symbol, followed by a reserved word identifying the specific form of the specification element. The specific form determines the specification element's structure, ending with the '**;**' symbol. Only program fragments may include embedded '**;**' symbols. The ';' symbol terminating the program fragment is determined by the nesting of '{' and '}' symbol pairs.

The forms of specification elements defined by the VIVA lexical and syntactic specification grammar are:

specification_element::=include_element
|program fragment
|start_element
|separator
|control_flow_function_element
|terminal_production
|nonterminal_production
|enumeration_declaration
|*global*_program_text

A detailed definition of each appears in the following sections:

3.0.0.10 Include Specification Elements

Include elements provide for the textual substitution of a file's contents at the point where the include element occurs in the input specification. Includes may be nested (include appears in an included file), however recursive

includes are disallowed. Since the same file can be know by many different aliases within a file system, recursive includes may not be detected in all cases.

include_element::=**#include** *path_and_file_name*_string_literal ‘**;**’

Within the include element, the string denotes the file to be included. If a fully qualified path is not given within the string, the specified file should be found in either the current working directory (first priority) or the directory in which the file containing this include element occurred, possibly using a qualified pathname (second priority).

3.0.0.11 Program Fragment Elements

Program fragments provide for inserting fragments of program code into the specification file. The fragments are inserted by textual substitution of the <language_dependent> section when subsequently referenced, by use of the identifier (declarator), later in the specification file. During textual substitution, the **#program**, declarator, optional language identifier, outer brackets and closing semicolon are elided.

program_fragment_element
::=**#program** *declarator*_**identifier**
[*language*_identifier] program_text ‘**;**’

program_text::= ‘**{**‘ <language dependent> ‘**}**’

The language identifier denotes the language used by the program fragment. By default, the language is CPlus-Plus (case insensitive); support for JAVA program fragments is expected. Other than matching brackets within the language dependent section while looking for the concluding outer bracket, the language dependent section may contain any constructs accepted by the compiler denoted by the programming language specifier.

For C++ program fragments, the language dependent section is inserted into a function having visibility to any parameters of the element into which it is inserted and any preceding declarations inserted by prior program fragments inserted into the same element. For example, program fragments inserted to provide an element’s strategy may include declarations used to form the element’s constraints.

Global program text is inserted at the top of the output source file. It logically succeeds the enumeration declarations.

3.0.0.12 Start Element

The start element identifier refers to one of the non-terminal elements at which language production should begin. In the absence of a start element, the first non-terminal element which is not used by any other non-terminal element becomes the start symbol. When the start element is inferred, tools processing the specification file(s) must clearly identify the inferred starting symbol.

start_element::=**#startidentifier ';'**

3.0.0.13 Separator Elements

Separator elements denote the set of characters which may be used to distinguish one lexical token from another under language generation conditions under which the absence of a separator character would result in a malformed language production. For example, two identifiers generated sequential without a separator character would result in a malformed language production resembling a single identifier. Anywhere one separator may be inserted, more than one separator may be inserted. In the absence of a separator element, the blank, horizontal tab, vertical tab and new line characters are used. It is a reported error if a specification contains more than one separator element denoting different separator character classes.<< this needs to be expanded to handle a full production, so comments can be considered as seperator. Also optional flow function to provide selection biasing for separator selection.>>

separator::=**#separator** character_class **';'**

3.0.0.14 Control Flow Function Elements

Control flow function elements define mechanisms determining control flow among either alternative language productions or iteration of a single language production. Actuals associated with formal parameters at a call site provide the function's evaluation context. The function's return value either denotes an alternative via logical position or enables continued iteration via a non-zero value.

Various alternatives are given a position starting with 1 assigned to the first alternative. If the alternatives are denoted by items and vertical bars ('|'), the first item is given position 1. If the alternatives are given by a character class, positions are associated with the underlying character set, not the order in which the characters appear in the character class. It is an error, reported at runtime, if the function returns a value denoting an undefined alternative.

control_flow_element::=**#function**
*declarator*_**identifier**
'(' [parameter_declaration { **','** parameter_declaration }] ')'
*return_type*_**identifier**
':'
{ *program_fragment*_**identifier** }
';'

parameter_declaration::=*type*_**identifier**
*parameter_declarator*_**identifier**
[**'='** *initial_value_program_fragment*_**identifier**]

The return_type_identifier, program_fragment_identifier (if any) and type_identifiers (if any) must be previously defined in a program fragment element. The language associated with the program fragment containing the type declarations and all program fragment identifiers must match. The return type identifier must denote a type convertible to an unsigned integer. Common return types include signed integers, enumerated types, and character types.

3.0.0.15 Enumeration Declarations

Enumeration declaration elements introduce enumeration declarations in the output source. One enumeration with the declarator "Language" is required before any language qualifier elements. The language specifier identifier in the language qualifier element must be a enumeration tag in the Language enumeration declaration.

enumeration_declaration::=**#enumeration**
*enumeration_declarator*_**identifier :**
*enumeration_tag*_**identifier** {, *enumeration_tag*_**identifier**}
;

3.0.0.16 Language Qualifier Elements

Language qualifier elements establish the dialect(s) of the language being tested associated with the next terminal or non-terminal element. A given terminal or non-terminal may be preceded by zero or more language qualifiers. In the absence of any language qualifier, the terminal or non-terminal is assumed to be valid for all language dialects. If one or more language qualifiers precede the terminal or non-terminal, the terminal or non-terminal may only occur in a language production if the language specifier matches one of the language specifiers active in the tool reading the VIVA language specification. More than one language specifier may be active at the same time.

Each language specifier includes one or more strings referencing the following terminal or non-terminal element back to the applicable language specification. The string's body takes the form of a document name, ':', and reference within the document. These strings are used to develop comments and other messages relating terminals or non-terminals back to the applicable language reference manuals.

language_qualifier::=**#language** *language_specifier*_**identifier**
string { **','** **string** }
';'

3.0.0.17 Association Lists

Association lists provide the bindings at a control flow function call site between actual values and formals of the control flow function. Association is strictly positional; the first actual is associated with the first formal parameter and so on.

association_list::=‘(‘ association_element { ‘,’ association_element } ‘)’

Actual values include (previously declared) identifiers denoting a value, integer literals, string literals, and references to previously declared program fragments. The program fragment must have the same source language as the control flow function associated with the call site (but may differ from that of the control flow function being called).

association_element::=**identifier**
|**integer**
|**string_literal**
|**‘{‘** *program_fragment_***identifier ‘}’**

3.0.0.18 Productions

Productions may be terminal (lexemes) or non-terminal (defined at least partially in terms of other productions).

production::= terminal_production
|nonterminal_production

3.0.0.19 Terminal Production

Terminal productions denote single lexemes to be emitted during language generation. A given terminal production within the VIVA specification may result in generation of many different lexemes during language generation, based on the inherited parameter values and subsequent evaluation of flow functions.

Each terminal production is declared by an identifier. This identifier is subsequently used in a non-terminal to initiate generation of a terminal production lexeme into the language production.

terminal_element::={ language_qualifier }
#terminal
*declarator_***identifier**
‘(‘ [parameter_declaration { **‘,’** parameter_declaration }] ‘)’
‘:’
terminal_item { terminal_item }
‘;’

Lexeme formation is defined by zero or more terminal items (the null lexeme is a degenerate case in which no characters are added to the language production). One or more terminal items may be encountered during pro-

duction. Each terminal production is in turn defined either as a list of one or more alternatives, an iterated terminal production, a character class, a specific character literal, a string or a program fragment inserted by reference.

terminal_item::=**‘(‘** terminal_item { ‘|’ terminal_item } **‘)’**
[**‘@’** *flow_function*_**identifier** association_list]
|character_class
[**‘@’** *flow_function*_**identifier** association_list]
|**character_literal**
|**string_literal**
|**‘<‘** *program*_**identifier** **‘>’**
/**program_text**

A list of alternatives, denoted by surrounding parenthesis and intervening vertical bars, may be followed by an ampersand (‘@’), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list denotes which alternative production occurs. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value within the domain of the alternative productions.

An iterator, denoted by surrounding parenthesis, may be followed by an ampersand (‘@’), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list either results in a 0 value denoting no further iterations or a non-zero value denoting continued iteration. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value allowing a random but gradually increasing number of iterations for each call to the flow function.

A character class, denoted by surrounding square brackets, may be followed by an ampersand (‘@’), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list denotes which character production occurs. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value within the domain of the character set productions.

The character and string literal result in the generator directly emitting the specified character or string into the language production. Although the character literal must be quoted in the specification, neither single nor double quotes appear in the emitted language production.

Finally, a program fragment may be inserted at the point where any terminal item may appear. No characters are added to the language production by the program fragment unless explicitly done by the program fragment.

3.0.0.20 Non-Terminal Production

Non-terminal productions denote lexemes sequences to be emitted during language generation. A given non-terminal production within the VIVA specification may result in generation of many different lexemes during language generation, based on the inherited parameter values and subsequent evaluation of flow functions.

Each non-terminal production is declared by an identifier. This identifier may be subsequently used by other non-terminals to initiate generation of a non-terminal production into the language production.

nonterminal_production::={ language_qualifier }
#nonterminal
*declarator*_**identifier**
‘(‘ [parameter_declaration { **‘,’** parameter_declaration }] ‘)’
‘:’
non_ terminal_item {non_ terminal_item }
‘;’
non_terminal_item::=**‘(‘** non_terminal_item { ‘|’ non_terminal_item } **‘)’**
[**‘@’** *flow_function*_**identifier** association_list]
|character_class
[**‘@’** *flow_function*_**identifier** association_list]
|nonterminal_production_**identifier** association_list
|terminal_production_**identifier** association_list
|**character_literal**
|**string_literal**
|**‘<‘** *program*_**identifier** ‘>’

/**program_text**

A list of alternatives, denoted by surrounding parenthesis and intervening vertical bars, may be followed by an ampersand (‘@’), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list denotes which alternative production occurs. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value within the domain of the alternative productions.

An iterator, denoted by surrounding parenthesis, may be followed by an ampersand (‘@’), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list either results in a 0 value denoting no further iterations or a non-zero value denoting continued iteration. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value allowing a random but gradually increasing number of iterations for each call to the flow function.

The mechanism that is used for a flow function depends on the enclosed structure. Note that alternative list flow functions return an integer that is a selection amongst the alternatives, while iterators return a binary selection (continue or stop). The alternative values are zero origin.

A character class, denoted by surrounding square brackets, may be followed by an ampersand ('@'), flow function identifier, and parameter association list. If a flow function clause is given, evaluation of the flow function in the context of the association list denotes which character production occurs. In the absence of a flow function clause the VIVA generation tool synthesizes a pseudo-random flow function producing a value within the domain of the character set productions. The selection is zero origin.

The character and string literal result in the generator directly emitting the specified character or string into the language production. Although the character literal must be quoted in the specification, neither single nor double quotes appear in the emitted language production.

Finally, a program fragment may be inserted at the point where any terminal item may appear. No characters are added to the language production by the program fragment unless explicitly done by the program fragment.<< need to specify how a fragment inserts characters into the language production>>

Specification Constraints

Although the VIVA lexical and syntactic grammar permits a unique parameter declaration for each terminal and non-terminal, for consistency FTL Systems has chosen to write all terminals and non-terminals with a common parameter declaration list (potentially augmented for a given production). This section describes the four different kinds of constraints which are propagated, as parameters, up and down the frontend specification grammar described in the Section (See “Lexical and Syntactic Specification Grammar” on page 39.):

Strategy Constraints (strategy_in, strategy_out),

Kind Constraints (kind_in, kind_out),

Type Constraints (subtype_in, subtype_out), and

Value Constraints (value_in, value_out).

In order to realize a common front-end specification behavior despite a variety of parameter passing mechanism using in the underlying implementation language, each assertion parameter occurs exactly twice in each front-end specification grammar rule (first as a parameter value passed into the rule, then as the possibly modified value returned). Constraints must appear in the order given above and all rules must have all constraints.

Placing the above constraints on all front-end specification grammars *does* result in many parameters being passed without usage by the test generator. However with the wide-spread availability of cut-and-paste editor functionality and even selective display masking of repetitive parts [1], the cost of generating and maintaining a canonical signature is minimal. With an effectively designed test generator implementation, parameters which are not actually used by the caller need not contribute to test generator overhead (assuming a global knowledge of all specification rules; the rule call tree). Having a single production signature for all rules eases the cognitive task of rule generation (less to go wrong) and eases design of interchangeable specifications and assertion / generator implementations (since specification need not utilize the same set of rules).

In a like fashion, repeating each assertion twice (once with “mode” in and once with “mode” out) addresses a wide variety of test generator implementation languages. Generally each rule in the specification translates into one or more subprogram calls in the test generator implementation. Some implementation languages, such as C++, permit subprogram (function) reference parameters. These parameters permit the callee to alter the call value so that the changes are visible to the caller on return. Sometimes this is useful, however at other times it is useful to have both the original call value as well as the return value available. Even with reference parameters, a single parameter requires development synchronization between the caller and callee so that any assumptions made by the caller concerning changes to the reference value remain true throughout the software maintenance cycle. Other implementation languages permit varying pass by reference mechanisms or lack such mechanism entirely. A sufficiently clever test generator creation tool can employ pass by reference of a single parameter (rather than two parameters) through visibility into the entire tree of rules and actions. The cost of this sophistication is *unlikely* to be worth the gain in test generation performance and thus is not contemplated at this time.

1. Such as Emacs’ outline mode employs.

Strategy Constraints

Strategy constraints guide a rule in the selection of correct or incorrect productions. There are two, largely orthogonal aspects to strategy constraints: the **certainty** with which an emitted production is in error and the **kind of error** which might be generated.

Since the certainty with which an error is generated has several mutually exclusive values, an enumeration may be used, VVIAStrategyCertainty:

enum **VIVAStrategyCertainty** {
IS_LEGAL,
MUST_BE_LEGAL,
MAY_BE_LEGAL,
MAY_BE_ERRONEOUS,
MAY_BE_IN_ERROR,
MUST_BE_ERRONEOUS,
MUST_BE_IN_ERROR,
IS_IN_ERROR };

Certainty values beginning with **MUST_** or **MAY_** are propagated downward in the rule hierarchy, whereas those beginning with **IS_** are propagated upward in the rule hierarchy, reflecting decisions which have already been made.

If an error is generated by a rule, many different kinds of errors may be permissible. In general these errors can be enumerated (as VIVAStrategyKind) but cannot be represented as mutually exclusive parameter values:

enum **VIVAStrategyKind {**
LEXICAL_ERROR,
SYNTAX_ERROR,
TYPE_ERROR,
SUBTYPE_ERROR,
AMBIGUOUS_OVERLOAD_ERROR,
NO_VALID_OVERLOAD_ERROR,
PROCESS_COMMUNICATION_ERROR;
SOLUTION_ERROR };

Combining the certainty and strategy into a single composite VIVAStrategy type yields the VIVAStrategy class:

class **VIVAStrategy**
{
public:
VIVAStrategyCertaintycertainty;

Boolean
is(**VIVAStrategyKind**v);
// Following methods have side-effects on VIVAKind value...
void
assert_is(**VIVAStrategyKind**v);
void
assert_is_not(**VIVAStrategyKind**v);

protected:
};

Kind Constraints

Kind constraints describe general characteristics of the rule production. Like strategy kinds, VIVAKind constraints can be enumerated, however a single rule parameter may match zero or more kind constraints. For example the expression being produced may be both an rval and a signal. Thus the same enumerated type and subsequent composite class used to represent strategies may be used for constraints:

enum **VIVAKinds** {
NO_ASSERTION,
IS_RVAL,
IS_LVAL,
IS_LITERAL,
IS_CONSTANT,
IS_VARIABLE,
IS_SIGNAL,
IS_GUARDED_SIGNAL,
IS_FILE,
IS_DYNAMIC_OBJECT,
IS_TERMINAL,
IS_QUANTITY,
IS_ENTITY,
IS_ARCHITECTURE,
IS_CONFIGURATION,
IS_PACKAGE,
IS_COMPONENT,
IS_BLOCK,
IS_PROCESS,
IS_LOOP,
IS_GENEATE,
IS_COMPONENT_CONFIGURATION,
IS_SCALAR_TYPE,

IS_DISCRETE_TYPE,
IS_INTEGER_TYPE,
IS_BOOLEAN_TYPE,
IS_BIT_VECTOR_TYPE,
IS_REAL_TYPE,
IS_TIME_TYPE,
IS_FILE_TYPE,
IS_FP_TYPE,
IS_NATURE,
IS_UNCONSTRAINED_ARRAY,
IS_CONSTRAINED_ARRAY,
IS_RECORD_TYPE,
IS_RECORD_NATURE,
IS_LOCALLY_STATIC,
IS_GLOBALLY_STATIC,
IS_NON_STATIC,
IS_UNIQUELY_DEFINED };

class VIVAKind
{
Boolean
is(**VIVAKinds**v);
void
assert_is(**VIVAKinds**v);
void
assert_is_not(**VIVAKinds**v);
protected:
}

Type Constraints

Type constraints represent a set of constraints on one or more types associated with a rule. Type constraints are represented as a list of AIRE IIR_TypeDefinition objects using a (new) IIR_TypeDefinitionList following other AIRE IIR lists. The absence of a type definition assertion is denoted by a NIL_IIR_TypeDefinition. See the AIRE IIR specification (http://www.ftlsystems.com) for further details.

Value Constraints

Value constraints represent a value produced by an expression production. Although values are not generally used for front-end testing, they are most readily included along with types. Values are represented as AIRE IIR_Literal, IIR_Expression and IIR_Declaration objects. The absence of an expression is denoted by a NIL_IIR value. See the AIRE IIR specification (http://www.ftlsystems.com) for further details.

CHAPTER 4

Sequential Extrinsic Functional Test Generation

Introduction

This chapter describes the functional test generator(FTG) that produces the executable tests' VIVA files. It explores the functional paradigm used to describe the execution semantics and how we map that semantic knowledge to generation of the specific tests. It also describes how to extend FTG to create more tests.

FTG uses the Advanced Intermediate Representation with Extensibility/Common Environment (AIRE/CE) as the starting point for its knowledge base. AIRE/CE classes are augmented or extended as needed to provide the knowledge necessary to create the tests. FTG generates tests for libraries. These libraries under test and the libraries referred to by the libraries under test must be analyzed into an AIRE/CE compliant analyzer that has been augmented by FTG. FTG generates executable tests for the libraries under test. If the library under test is the standard library, then additional tests are created to cover all language constructs that have execution semantics that are not covered by the VIVA Liveliness tests.

Each language construct has a set of FTG rules that describe to FTG how to create tests for that language construct. These rules are a combination of C++ classes and C preprocessor defines that allows for easy addition of more rules. A rule may invoke a series of other rules. Each test is added to AIRE/CE in a test library. Several non error tests may be combined into one test. The test library is fleshed out with the necessary language constructs to provide a complete executable description.

Data objects are the usual language construct used to detect proper execution. Data objects are a language construct that are described by a tuple of value, name, type, and kind. Data objects change value as the result of execution of expressions and statements. The type of data objects are chosen from the types defined by the type system. It includes test defined types as well as predefined types. The kind of object is selected from variable, shared variable, or signal. The FTG creates objects and values to test the modification of values by expressions and statements. It examines standard subprogram and process bodies to create input data that tests all paths.

The other major language construct used is assert. Assert is used to state conditions about data objects, or to identify execution paths that should not be executed.

FTG does not test basic computer function. For example, we do not claim to test that the adder of a machine is correct, we only test that the addition operation does add.

Type System

FTG will generate a series of tests for each type described in the knowledge base and contained in the target library.

There are three kinds of scalar types in VHDL. There are enumerations(Bit), ordinal (Integer and Time) and arithmetic (Real). Each of these require a different treatment for coverage of their uses. Aggregate types (array and record) are treated as a collection of the scalar types for test values. Access types are treated as a mode on the types they access.

Each scalar variable test has an equivalent test using an access to the types involved. For instance if we are testing an array index type, then we should repeat that test with an access to array.

FTG will create tests to ensure bounds checking on assignment and array indexing.

Type attributes are checked for each type. Error tests include boundary tests like T'HIGH'SUCC.

4.0.0.1 Enumeration value selection

FTG will exhaustively check an enumeration type of X elements when X is less than a run time defined limit. That is each binary operation will have 2^X tests. For enumerations that are too large to exhaustively test, FTG will cover the output domain of operations and use principles like transitivity of '\' to reduce the number of tests.

If an enumeration type is a subtype of a larger set of enumerations, tests will be created to ensure that the assignment of enumeration values outside the subset are error.

Package standard will also create a subtype of CHARACTER to test subtypes and resolution functions.

Ordinal value selection

Ordinal values can be mapped to integers, and arithmetic operations are valid. VHDL integer and physical types are ordinal.

- Operations

  FTG checks ordinal operations with 5 values. 'LOW - 1, 'LOW, random middle value, 'HIGH, 'HIGH +1. Operations will produce errors in some of these cases. Error tests need to check for disqualifying conditions like if a type wraps ('HIGH+1 = 'LOW), then var:= 'HIGH + 'HIGH will not result in an error.

- Indices

  Ordinal values as indices should have the same 5 values as operations, but also may have an additional test structured around induction.

Package standard will create a small integer type to test range checking.

Arithmetic value selection

Arithmetic values approximate real numbers. VHDL float is an example. As they are an approximation, exact specification of the results of operations is not possible. Results tested to be within a range.

Composite value selection

Composite values are built up from the value selection for their elements.

4.0.0.2 Array value selection

The value of array elements will be taken from the type of the element and the operation to be tested. Also, there will be array tests that revolve around the values for the index.

4.0.0.3 String value selection

Strings are one dimensional arrays indexed by a natural integer index whose element type is a enumeration whose tags are all one character. Strings have the same value selection criteria as arrays.

4.0.0.4 Record value selection

Record values will be a cross product of the test values for the types of the record elements.

Objects

Each type is tested with both variable and signal objects. An access to an object of non access type is tested as well. The tests are repeated with the different kind of objects. Variable objects may be tested in the same process, while signal objects are tested by another process.

Subprograms--operations, functions, and procedures

FTG tries to cover the outputs of a subprogram based on the type. It creates a set of desired output states. It backtracks through the statements of a subprogram creating a set of constraints that eventually are reflected to inputs or shared variables. These constraints are used to select values. (i.e. we have a simple function that returns the sum of inputs A and B. For a given desired output value X, A gets the constraint of X-B and B gets the constraint of X-A. A and B are selected from these constraints and their types. In a not fully constrained system like this, a when a random value is chosen, the most constrained is randomly selected and the set of constraints is updated.)

FTG tries to cover all execution paths through a subprogram. It does this by analyzing the control flow and creating input data to exercise the pathways. If some code is unreachable, it will print out a diagnostic message. Some paths may not be coverable, because of mutual exclusion.

Processes and Concurrent

Processes are analyzed similar to subprograms. Input/Output signals and shared variables are detected from expression reference and assignment statements within a process. Input values over time are constructed as needed to set the internal state of a process to create the proper traversals and output values. Several paths through a process may be covered by one test, if several invocations are needed to create the proper internal state.

Processes are created to evaluate expected signal changes.

Alternate Forms

Where there are alternate syntax forms to represent the same execution, then a test will be generated that tests each alternate form. A test of a concurrent signal assignment will also test the equivalent process.

Waveforms

Standard components with state will require a sequence of input signal values to cover the sequential paths and values of output. The same process used to create a sequence of calls for a subprogram with state is used for components with state.

Weaknesses

There are several weaknesses to this body of tests and any VHDL implementation should supplement this validation suite with executable tests of their own.

- Machine dependencies
- Undefined behaviors
- Non portable constructs
- Capacity
- Optimizations

Deliverables

The FTG source: a set of C++ files and a Makefile that compile into an AIRE application.

Documentation on how to add new rules to FTG.

VIVA grammar files for VHDL package standard which includes testing of the base language.

Future

VHDL-AMS

FTG Rules

FTG rules are the basis of the system. New rules can be added to create tests or add objects to the AIRE database for other rules. The effect of rules will be different depending on where they are placed in the structure of FTG.

FTG program structure

A rule falls into one of several categories. The categories correspond to one of the test areas: type, sequential statements, concurrent statements,...====more===. The category order is important, as the rule categories are processed in order

- type rules

  Type rules create new types and subtypes.
- sequential rules
- concurrent rules

FTG rule creation

New tests are added to FTG by adding rules. A rule is a C++ class derived from FTG_Rule, and a rule definition. Rules are defined to FTG in several .rul files. These are:

- ftg_type.rul
- ftg_seq.rul
- ftg_conc.rul
- more to come

Rule definition syntax is RULE(rule_category,rule_name,ir_object_kind, test_library_name, ...). The rule category is a check on the file it is included in (type rules in the ftg_type.rul file...). The rule_name is the name of the C++ class that implements the rule. ir_object_kind is the IR_Kind of AIRE/CE object that this rule applies to. The test_library_name is the library to find AIRE/CE objects to pass to the rule. If the library name is “ftg_all”, then the rule will be applied to objects in all libraries. If the library name is “ftg_any” then the rule will be applied to all libraries under test. These special library names allow a common rule set for many libraries.

```
class FTG_Rule {
public:
   virtual IR_Int32 invoke(IIR *iir_object)=0;
};
```

Rules are invoked for each object in the test_library_name of the iir_object_kind, if test_library_name is one of the libraries under test. Each rule will be invoked with a null object first. This will allow the creation of static data or other initialization, or the addition of objects to the database.

Rules may add new objects to the FTG_adjunct library, or to either the FTG_good or FTG_error libraries.

CHAPTER 5

VIVA Installation Guide

1. Make sure that perl is in the path
2. Place the following files in the perl directory. The directory is usually on UNIX /usr/local/lib/perl5 or something like that.

   Register.pm

   Viva_msg.pm

   VIVA_comm.pm
3. Place the following file in the home directory.

   TestGen.pl

   TUT.pl

   DemoWin.pl

   viva.rc

   viva_(platform)_tut.rc   (i.e. viva_solaris_tut.rc)

   viva_sparc
4. Edit TestGen.pl parameters at the top of the file, under the TestGen Wrapper Input Interface comment. Change $site_wrapper to the directory where TUT.pl is located. Change the blaa bla in the following statement open($FH_viva_config, “blaa blaa/viva.rc”) to a path where viva.rc is located. Also, change $TG_exec and $graphics to the same directory.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

CHAPTER 6

Annotated Bibliography

Robert S. Boyer and J. Strother Moore, **A Computational Logic Handbook**, Perspectives in Computing, Academic Press, 1988 (ISBN 0-12-122952-1).

C.A.R. Hoare and J.C. Shepherdson, **Mathematical Logic and Programming Languages**, International Serices in Computer Science, Prentice/Hall International (ISBN 0-13-561465-1).

Paul C. Jorgensen, **Software Testing: A Craftsman's Approach**, CRC Press (ISBN 0-8493-7345-X).

Matthias Gulbins and Bernd Straube, **Applying Behavioral Level Test Generation to High-Level Design Validation**, In Proceedings of the European Design and Test Conference, 1996.

Robert M. Poston, **Automating Specification-Based Software Testing**, IEEE Computer Society Press, ISBN 0-8186-7531-4.