ANT - A Test Harness for the NELSIS CAD System*

C. A. Schot, M. N. Sim & P. M. Kist

Delft Institute for Microelectronics & Submicronotechnology (DIMES)
Feldmannweg 17, 2600 GB Delft

&

Network Theory Section / Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft

Delft University of Technology, The Netherlands

ABSTRACT

ANT is a test harness for validation of a suite of design tools incorporated in a framework. Unique is that the harness is built upon the same framework used to support the tools and tests are modeled as hierarchical CAD objects. To our knowledge, there are no other harnesses for testing systems of this category, therefore the design considerations presented may provide an insight for others attempting validation at this level.

1. Introduction

Test Automation

An estimation published in [1] puts the price tag of testing at 50% of the cost of system development, thus automating the test process is an important way to slash costs. Test automation may be split into three categories [2]:

1) Automating the administrative side of the activity, e.g. looking after printing and the identification of tests, storage of sessions etc.

2) Automating the mechanical side of the activity, e.g. the running of the software under test within a test harness, the application of test data to it and the comparison of actual results with those expected.

3) Automating the generation of test cases.

In this paper we describe a test environment that covers the first two categories. The last category is considered the most difficult to automate and a problem yet to be tackled. However, before attempting automatic test case generation, it is essential to have a good test harness and this we have accomplished.

The investment of resources in test automation appears worthwhile because the evolution of mature frameworks and tools for VLSI CAD is an incremental process. New releases deviate only slightly from their predecessors, since the need for upward compatibility implies that the system is stable. However, even if altered only slightly, due to the complex interaction between sub-systems, one is often compelled to test the system as a whole. Moreover, the validation must usually be performed on several machine types.

As far as we know the state-of-the-art CAD systems do not have a test harness. For example, the testing of frameworks within the JESSI CAD Frame evaluation sub-project has been carried out manually [3]. Automating the testing has several advantages:

1) Quality: since the testing becomes much cheaper in terms of man-hours, circuits of relatively larger order (which consume more time) can be used as inputs for test. This increases the quality of the test, since many problems in CAD software do not occur when small circuits are used as test data.

2) Reliability: manual testing is error prone due to the longevity and complexity of certain operations.

3) Quantity: the amount of tests can be increased, thus improving statistically the soundness of the test.

4) Replication: the same test can be repeated accurately, which enables a complete reconstruction of the whole design environment, up to the point where an error occurred.

5) Measuring: important run-time data can be gathered and stored, for analysis of performance, such as memory and CPU usage and tool execution time.

Design Philosophy

The design of our test harness is based on two principle concepts:

1) A test harness should be an integral component of the CAD framework.

2) The test data should be modeled as CAD objects.

This has the following advantages:

1) By building on top of the framework itself, a powerful test environment may be produced with minimal implementation effort.

2) The resources used to operate the system (Disk and RAM) are minimal.

* This research was supported in part by the commission of EC under ESPRIT contract 5082.
3) The (framework) client may use the harness for local testing.
4) The framework may be shipped with a test package which enables on-site testing.
5) Modeling tests as CAD objects on the same framework improves the learning curve of the harness for the client.
6) A formal channel exists for clients to communicate problems to the vendor, since a complete test suite may be defined as CAD objects which may be run at another site.

Structure of this Paper
Chapter 2 places the test harness within the broader scope of verification and validation. Chapter 3 gives an overview of the NELSIS CAD system. Chapter 4 presents design considerations related to the test harness. In chapters 5 and 6 we present the design and implementation respectively and conclude in chapter 7 with results, suggestions and future work.

2. Testing Concepts

Producing reliable systems requires a well planned, comprehensive application of several techniques by many players throughout the development life-cycle, collectively known as Verification & Validation (V&V) [4]. Verification involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous stage. Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements. The Test phase is applied to both verification and validation. Testing involves several tasks [5]:
1) Build the test harness; A driver is typically required.
2) Generate the inputs.
3) Determine the expected output, known as the test oracle problem.
4) Execute the module under test on the selected inputs, monitoring system/module behavior.
5) Compare the actual outputs to the expected outputs.

We focus on a specific type of test within the validation category known as System Testing which implies exercising the system as whole. The basic entity in system testing is the test case which is a collection of input data, output data, comparative data, and a sequence of functions to be executed. A combination of test cases to be executed in sequence forms a test suite. System testing implies driving and analyzing a (partial) set of test suites. There are two basic types of system testing: white-box, in which the test data is related to the structure of the program, and black-box (or functional), in which the test data is constructed from the specifications. The former is effective for probing for weaknesses in sensitive regions and suitable only for the system vendor, while the latter requires no knowledge of the internal structure and therefore is also suitable for the client.

3. The NELSIS (VLSI) CAD System

The NELSIS CAD Framework
The NELSIS CAD Framework [6] maintains information about the design and the design activities, so-called meta data, rather than operating at the level of the detailed designed descriptions. This involves primarily information on the presence of cells or design objects, their relationships, their version history, and the operations that have been performed on them. Design objects are categorized in view-types (e.g. layout, circuit). The framework does not make assumptions about the actual detailed design descriptions contained in the design objects, and therefore it can be used for CAD environments. The raw design data is operated upon by the design tools. The data meta is collected by the framework while the tools communicate with it to obtain access to the actual design descriptions. One of the major organizing principles in the NELSIS framework is the project. A project provides a local context for the design activities of one or more designers. It contains a collection of design descriptions that can be operated upon from within the project context.

The framework employs a CheckOut/CheckIn transaction model to support design transactions performed by tools on design objects. The meta-data management module (MDM) is used to administer the in-progress transactions and maintain a transaction history of completed transactions. Individual versions are administered with the MDM, along with such attributes as version-number or version-status. The framework layer interfaces to design tools via the data management interface (DMI), which hides many framework aspects from the tools.

The framework has a number of useful browsers for browsing through the meta data, namely hierarchy, version and data schema browsers.

The NELSIS Design Environment
A set of IC design tools is integrated on top of the NELSIS CAD Framework. These tools support the development of a wide range of IC's: from analog standard cell up to analog and digital full custom VLSI.

The tools are classified into three packages:
1) The Simulator package, including the digital mixed level simulator SL5, the well known analog simulator SPICE, graphical user interfaces and interfaces with standard network descriptions (i.e. EDIF).
2) The Layout package, including the interactive layout tool DALI, tools for layout generation and interfaces with standard layout descriptions (i.e. GDSII).
3) The Verification package, including layout rule verification, the extractor SPACE, and the layout-to-circuit verification tool MATCH.

Apart from these kernel packages a number of other specialized IC design tools are available.
4. Design Considerations

Basic Problems & Solutions

In chapter 2 a list of testing tasks is presented, the first being "build the test harness": but how? We concluded that the architecture of the harness depends on the type and quantity of test data to be managed. We studied the list of additional tasks and from [5] it was clear that the most critical testing task is item 3, i.e. defining the expected output. Apart from being the most time consuming task, the reliability of the testing is directly correlated. Our solution was based on the resources at our disposal, namely several fairly large VLSI designs, completed and simulated for correctness and some also fabricated and tested. Since these designs were home built with the NELSIS CAD System, we had access to all ancillary data, such as simulation files and complete design databases. This data represents many man years of painstaking work and is sufficiently reliable. Thus the domain of the output was chosen to be one or more NELSIS design databases together with related ancillary data in the form of files (in the UNIX file system) which we label an environment.

Having defined the expected output, the input definition (task 2) was decided to be in the domain of the output, thus the concept of a state machine is formed which is essential for creating test suites. Executing the model under test (task 4) is therefore performing part of a design trajectory. Thus, a test case consists of a pre-environment, a design trajectory and a post-environment, and a state is the set of environments that exist in steady-state (between transitions).

Specific Requirements

1) It should be possible to reconstruct an environment. This is justified by the fact that a test person should be able to manually examine any environment created during the execution of a test suite, after the test has been driven, in order to obtain additional information (if necessary). Test cases are extremely time consuming (e.g. simulation) and therefore it is not acceptable to start a test suite from scratch.

2) It should be possible to duplicate an environment. A pre-environment may be the input to several test cases. Since a test run causes a state transition, the harness must be able to save a pre-environment for re-use. We studied the problem and concluded that Nelsis design projects may be duplicated and relocated without loss of functionality or data. If this were not possible, our test method would not be practical.

3) It should be possible to initialize an environment. Often an initialization needs to be performed on a pre-environment to make it more suitable for a particular test run.

4) Tools for inspecting and re-setting environments and results, defining test cases etc. should be present.

We demonstrate how the environment mechanism is related to an automatic test-suite execution in figure 1. The rectangles and circles denote states and transitions (test runs) respectively. At the bottom left we have environment E1 which is the initial state. When test run

Figure 1. A test suite mapped to states and transitions

T1 has passed, environment E1 is transformed into E2. Because E2 is the pre-environment for two tests, T2 and T3, the environment must be sustained (rectangle denoted by S). When the tests T2 and T3 have passed, two new environments are created, E3 and E4. At this point the sustained environment may be erased. The two post-environments of T2 and T3 together form the pre-environment for test T4. When a test fails the (incomplete) post-environment is sustained by default. This is denoted in the figure by the dashed lines, leading to the state called non-complete.

5. Design of the Test Harness

The Automatic NELSIS Tester (ANT) consists of a set of test tools built on top of the NELSIS framework which enables the design and management of test cases, and their automatic execution. Test cases are mapped to design objects (called test-objects for compliancy). Thus, in analogy, a test suite may be mapped to a hierarchical object. Using the frameworks design system interface the user can browse through the test hierarchy to examine and manipulate the status of the tests. A special tool provides the user with a graphical interface to the ANT system, which enables easy invocation of the test tools with the necessary parameters and displays essential actions on a monitor.

The test harness manipulates a pilot project which contains all information concerning the design and execution of tests. The test data consists of (VLSI) design data and related design projects. Test cases are defined by the user in a format that can be processed and stored in the pilot project. ANT executes a test suite by traversing the hierarchy. It gathers information on the way, prompts the user for additional information if necessary and reports failed tests. It offers various tools to rectify tests, reset test sequences, and restart execution.

Figure 2 illustrates a simplified test suite. The top level object, T_Nelsis, represents the testing of the whole CAD system. It splits into two objects, T_Frame and T_Tools. These represent different test groups: the first being common framework aspects and the latter being design tools.

Under T_Frame are tests for the user interface and the
Design and Test Modes

The user may use the harness in two working modes: design and test, as shown in figure 3. In plate A, each circle represents a major phase in the design cycle (as most commonly carried out). The major operations performed in the particular phase are shown on the left, and the tools used during that phase are presented on the right.

In plate B, each circle represents a major phase in the test cycle, the associated major operations (left) and the tools involved (right). This tools will be discussed further on in chapter 6.

The Design Cycle

A test definition is constructed of the initialization, execution and checking components. The definition is not likely to be correct after the first installation and execution, it is usually a cyclic and iterative process. During design, each phase may be checked separately. When developing a test-object that is hierarchically dependent on one test-objects, the pre-environment may be sustained so that the test can be reset quickly.

The Test Cycle

During the test cycle there are two test options. The first is executing each phase separately which enables checking each phase in detail, and is usually used during

Figure 2. A simple test suite mapped to hierarchical CAD objects

browsers. Under T_TOOLS are the tests for the tool space. The tool execution sequence occurs in the opposite direction of the hierarchical representation. The lower level test objects are the son tests of the higher test objects (the fathers). When a father has two or more sons, it implies that the son tests must have passed their test before the father test can be executed. In the figure, the test T_browsers cannot be executed before the tests T_HBrowser and T_HBrowser have passed. If a test object has multiple fathers, ANT creates a duplicate environment, which enables both fathers to execute without loss of consistency. Thus, if an environment created by the execution of I_space is used by T_space_f, then a copy of the environment is made for the execution of T_space_h. Once the test has progressed to a higher level, ANT may destroy environments of a lower level, to avoid data explosion e.g. once T_space_h and T_space_f have passed, their pre-environment may be destroyed, i.e. I_space f post-environment.

Data Schema

The data schema is defined according to the OTO-D model [6]. Rectangles denote types. A line from the bottom edge of a rectangle to the top edge of a rectangle implies that the bottom type is an attribute of the top type. A line from the bottom corner of a rectangle to the
phase, the complexity of a test object is broken down into a specialization of the bottom type (inheritance).

**ViewType:** There are five ViewTypes. During the design phase, the complexity of a test object is broken down into separate sections, each represented by a ViewType.

1) **INIT** consists of DO's that contain initialization data: input files and initialization scripts.

2) **EXEC** consists of DO's that contain execution data: names of expected output files and execution scripts.

3) **CHECK** consists of DO's that contain check data: reference files and check scripts.

4) **TO** consists of DO's that contain organization data: report scripts and documentation.

5) **TEST** consists of DO's that contain result data that is gathered during the execution of the test phases.

**Hierarchy:** Hierarchy only applies to the TO ViewType. The Hierarchy registers the father-son relationships between test objects.

**Equivalence:** The ANT system maintains two types of equivalence, therefore Info may have two values. The first represents the normal NELSIS equivalence, i.e. one DO (DesignObject) is derived from the another DO through a tool manipulation. The latter equivalence is used to model a test object set. Each equivalence instance relates DO's from different ViewTypes that together form an entity.

**DesignObject:** The DO has the attributes: Vnumber, Vstatus, Module, Imported and Date. Vnumber is the version of the DO. Date contains the test installation date. Module is a composite attribute containing the attributes: LastVersion, Name, ViewType, Project, TOEnv and Designer. Last Version contains the last installed version of the test. Name is the name of the test. TOEnv is only meaningful for DO's of the TO ViewType. It may have the values: Saved, NotSaved. Saved implies that the post-environment of the test object has been saved.

**Import, Export and Library** are handles to test objects in other pilot projects. The Import mechanism is not meaningful for the TEST ViewType. It is used to (hierarchically) import a complete test object.

**Project:** Type *Project* is used in two contexts, the normal attribute of *Export*, and the additional attribute of *Module*. The latter associates a design project with the test object.

**Vnumber & Vstatus:** In the TO ViewType the Vnumber (version number) enables defining many test objects of the same name. The meaningful Vstatus (version status) values are *Actual*, *Working* & *Backup*. When the ANT system is manipulated in the Design mode, the Working version is selected by default. When in the test mode, the Actual version is selected.

In the Design category the versioning mechanism is the same as in the Framework. In the Test category, all objects are of the Derived status, since they are the result of extraction based on the Design ViewTypes.

**Transaction:** Transaction registers the tool interactions with the database.

**The ANT Tools**

Figure 5 shows how the tools are incorporated in the framework.

Dog-eared rectangles represent files. Rectangles represent directories. Circles represent ANT tools. In the center, the five rectangles represent the ViewTypes of the pilot project. The arrows show the direction of data flow during tool execution. Arrows connected to the large rectangle that engulfs all the ViewTypes imply that the data may flow to or from several ViewTypes. A stipple lined arrow implies that a tool manipulates the data in the ViewType pointed to, but does not necessarily transport it (e.g. rmTO).

ANT provides three categories of tools: General purpose, Design and Test.

The general purpose tools are antui, mkPilot, rmPilot, tsui, impTO and mkTO.

The first five tools are not shown in the figure. They include test pilot project creation and deletion tools and user interfaces. With impTO test objects from other test pilot projects can be included (called). mkTO installs test objects in the database.

The test tools, envTO, execTO, checkTO, reportTO and testTO, are used for (automatic) test suite execution. These tools control the environment of tests (Save, (Re)Set), perform test runs (in a hierarchical order), check the results and generate the test report.

The design tools are extra tools needed for the definition of the test cases and include the tools defTO, defSHS (define SHell Script), defNAS (define Inter Active Script), dbEdit and rmTO.

They extract test object definitions, control (interactive) test scripts from the database for editing or long term storage, or (in case of new data) provide template definitions. rmTO removes test objects from the system.

A typical test runs as follows:

During the install phase the tool *mkTO* parses the (possibly hierarchical) test object definition script and
stores the test cases in the streams organized in the
different ViewTypes of the pilot project. envTO invoked
in mode SET, creates the initial state for each test in the
workspace, a directory in the pilot, in which the actual
test run takes place. execTO, guided by the information in
the EXEC ViewType, performs the actual automatic or
interactive test run. Results are stored in the TEST
ViewType. checkTO, guided by the information in the
CHECK ViewType, compares referential data from the
CHECK ViewType against the test results. Then
reportTO may generate a report.

After each test, the data in the workspace, or post-
environment image, may be saved in the TEST
ViewType using the tool envTO in SAVE mode. The
image can be recaptured (if so desired) using envTO in
RESTORE mode. When envTO is invoked with the
RESET mode, the pre-environment of the test is restored
by copying the post-environments of all its sons from the
TEST ViewType to the workspace.

7. Conclusion
   Results
At the moment ANT is available only for internal use
only at the university where it is being tested and
improved. We use ANT for regression testing of the
NELSIS Framework and tool releases and for reporting
problems back to the framework development group.
The harness is further used as a basis for two other
environments built on top of the NELSIS Framework:
1) OPI [7] , a rapid prototyping environment, which uses
the harness for module testing.
2) NTE [8] , an integrated training system for NELSIS
users in which test cases are replaced by training
sessions.

Suggestions
In our opinion framework consumers should demand a
test harness together with an elementary test suite to be
shipped with the framework (system). Customers
studying the extent of the suite and executing it, may gain
valuable insight into the reliability (and functionality) of
the system without performing extensive in-house
evaluation. Moreover, much better feedback of problems
is possible.

Such a harness also enables formal communication
between the development and the validation group. This
is extremely important, since a key element to the
effectiveness of testing is that a program be observable
and controllable [9] , which results in a program being
easily testable. These characteristics can be introduced
by the development team only.

Future work
The automation of interactive mouse driven graphical
tools is a major problem. At the present, ANT lacks such
a mechanism and pops up communication windows that
guide the user (in this case manual test execution).
This problem has been solved in [10] by utilizing the X
record and playback monitor XTM. Being a public
domain tool we believe this to be a feasible solution to
our problem.

The second problem is the development of test cases.
Although ANT provides support for this procedure,
bewide being a time consuming process it is primarily a
theoretical problem. Only raw estimations have been
made about the test coverage of the test cases, which
influences the soundness of the system test. Several
students work on this important but still barren area [2].

References
1. B. Korel, “Automated Software Test Data Generation,”
   870-879 (May 1990).
2. M. Ould, “Testing - a challenge to method and tool
developers,” Software Engineering Journal, pp. 59-64
   (March 1991).
3. J. Heaton, “Evaluation Sub-project Annex to Partner
   Reports to Start-up Phase Report,” JCP/ICL/001-01,
   (June 1991).
4. D. Wallace and R. Fujii, “Software Verification and
   Validation: An Overview,” IEEE Software, pp. 10-17
   (May 1989).
5. D.M. Hoffman and P. Strooper, “Automated Module
   Testing in PROLOG,” IEEE Transactions on Software
6. P. Ringley and P. van der Wolf, “A Design Platform for
   the Nelsis CAD Framework,” 27th ACM/IEEE Design
7. M. Sim, K. Schot, and P. Kist, “Configuration
   Management in an Object Oriented Persistent
   Programming Environment for CAD,” ACM/IEEE
8. K. Schot, M. Sim, and P. Kist, “A CAD System for
   VLSI Design Training,” Second Eurochip Workshop on
9. R. Freedman, “Testability of Software Components,”
   IEEE Transactions on Software Engineering, Vol. 17,