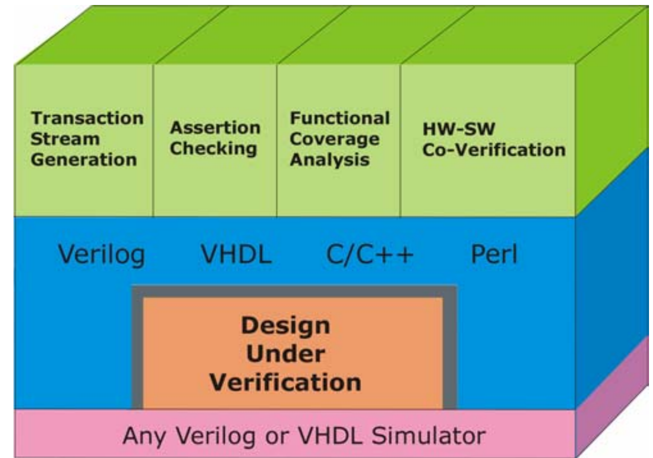
# TestWizard • Testbench Automation

***The Power of Advanced Verification and Simplicity of Verilog and VHDL***

## HIGHLIGHTS

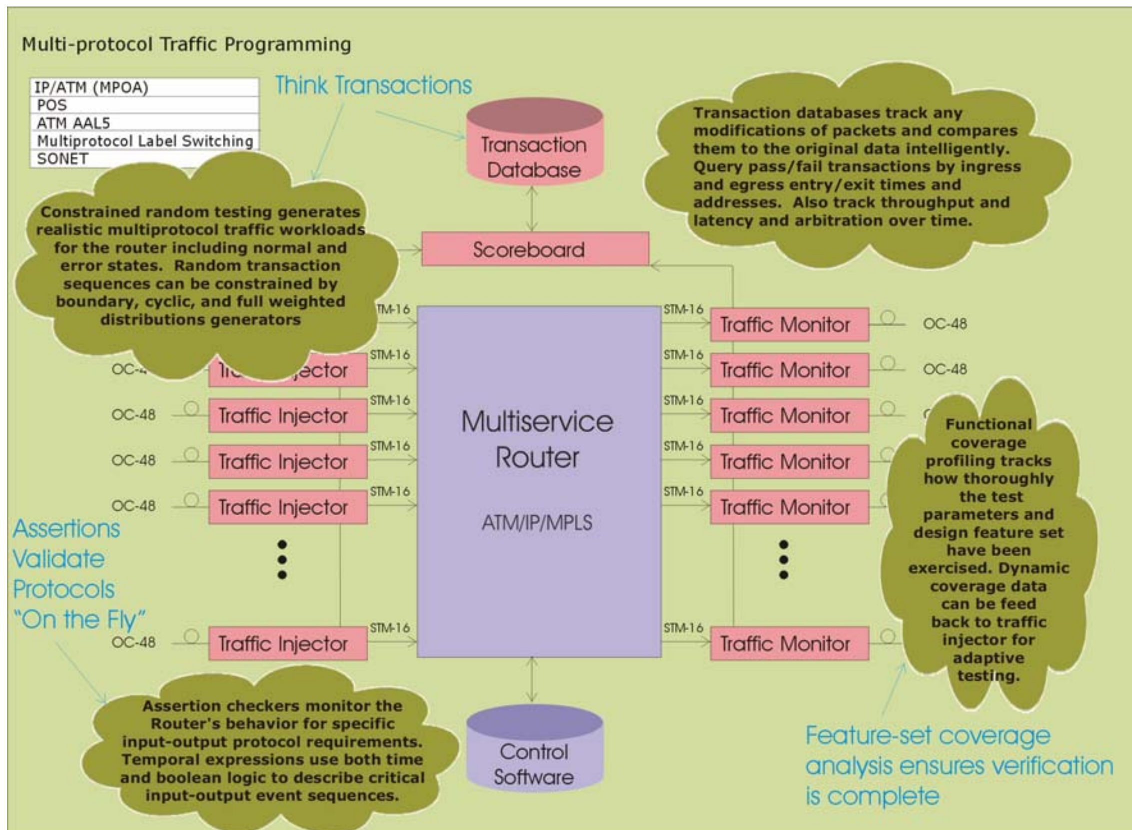
- Single system for transaction level verification and simulation
- 3X better productivity using advanced Verilog HDL, VHDL, and C-based testbench automation
- Supports all leading Verilog and VHDL simulators
- Algorithmic test generation using intelligent random scenarios
- Coverage analysis grades tests versus design architecture
- PSL-based assertion checking validates physical and system-level interface protocols
- Simplifies testbench reuse across projects and tools
- Evolutionary approach that fits in existing verification
- Faster debugging using transaction-level abstraction
- No new languages




## OVERVIEW

TestWizard is a transaction-based verification test development system. Tests can be written in Verilog, VHDL, or C/C++, eliminating the need to learn a new language. TestWizard works with all popular Verilog simulators and has constructs that augment Verilog and VHDL environments to speed development of directed or random testbenches. TestWizard adds feature-level coverage analysis to grade tests versus design architecture to ensure the integrity of the functional verification. TestWizard also adds assertion checking to validate complex temporal properties of physical and system interfaces.

All together TestWizard simplifies the development, maintenance, and reuse of concurrent, adaptive, directed, and random tests. TestWizard raises verification from the signal level to the higher and more productive transaction level while enabling debug at the signal level when necessary. Design and verification engineers can now focus on the overall functionality of the design, instead of low level details of smaller portions of the design.




A fundamental advantage of TestWizard is that users do not have to learn new languages or work in a multi-language environment. This promotes verification reuse and means that all members of the design and verification teams have the knowledge to solve verification problems.

## THINK TRANSACTIONS

Traditional verification methods of HDL-based testing and waveform analysis are no longer sufficient to handle the increasing levels of architectural complexity of SOCs and multi-chip systems. Today's system designs require thorough and rigorous functional

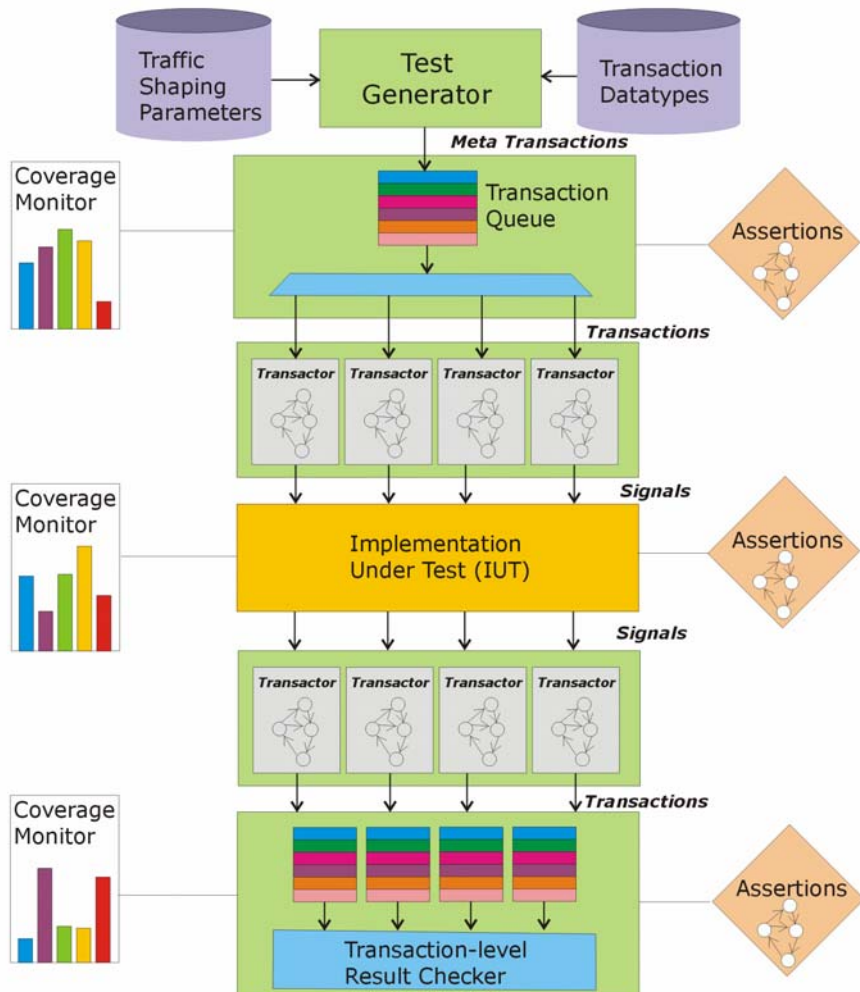
verification testing to verify system conformance of hardware-to-software interfaces, signal processing and multimedia algorithms, and communications and bus interface protocols. To keep up designers need to elevate verification to the next level of abstraction - transactions - and utilize transaction-based testbench automation tools to capitalize on a better verification strategy making it far easier to create tests, reuse code, debug simulations, and assess functional verification coverage metrics.

A transaction represents the transfer of data and control over a system interface to perform some basic system operation. A system interface is the collection of signals, busses, and clocks and an exact protocol specifying how the data and control flow. Transactions can be as simple as sending an Ethernet packet or as complex as passing Packet Over SONET (POS) packets through a MPLS enabled multi-service router design. Prior to TestWizard, designers generated sequences of transactions manually, and recorded transactions in log files, and searched through the log files to understand and debug simulation - a very inefficient approach. TestWizard greatly simplifies generating transactions and verifying output transactions for content and relationships to other transactions as well as internal system operation and sequencing.

## MODELLING TRANSACTIONS IN HDLs

Avery has developed powerful extensions to Verilog and VHDL to support modeling transactions and test sequences called the Verification Language Extensions (VLE). VLE captures the power of high level programming techniques, together with the simplicity of standard hardware description languages. These extensions implement datatypes and functions explicitly designed to easily represent complex transactions. The VLE includes constructs for:

- User-defined records - provides high-level datatypes to represent complex transaction datastructures
- Record streaming simplifies transaction-to-signal level BFM interfaces
- Random case generation creates key test scenarios easily
- Lists – multi-level, mutable lists enumerate value sets of all types
- Semaphores - shared HDL/C control to shared resources
- Signal history - provides runtime access to value change history of HDL variables
- Record history - provides runtime access to record mailboxes
- Transaction database - provides runtime database query to mixed transaction record history




*TestWizard is a comprehensive transaction-level verification environment integrated with your Verilog or VHDL simulator*

Relying on an approach to decouple test content from signal-level issues, TestWizard builds upon bus-functional model (BFM) techniques that are commonly used in verifying complex systems. BFMs or transactors convert transactions to the lower level signals at the interface of the design under verification. With TestWizard, the tasks of developing, running, diagnosing, maintaining, and re-using transactors and verification test suites become much easier.

## NO NEW LANGUAGES

TestWizard provides functions to ease test development. It allows users to write tests in Verilog, VHDL, C/C++, and Perl while using TestWizard functions specifically designed to make development of transaction-based test generators quick and easy.

## MULTI-LANGUAGE BASED TESTING AND HW-SW CO-VERIFICATION

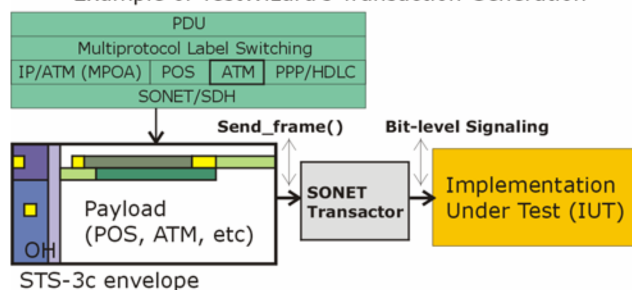
Today's diverse system-level verification environments are comprised of HDL simulators, hardware emulators, embedded and application software, and IP models written in Verilog, VHDL, and C/C++. TestWizard seamlessly integrates into this environment through VCI, a multi-agent collaborative infrastructure.

The VCI API supports the use of C/C++ and Perl for high-level transactional test development as well as integration of IP models and system software modules. The VCI supports robust synchronization and concurrency options involving multiple processes which ensures timing accuracy, repeatability, and determinism.

IP models developed in C/C++ can be easily interfaced to HDL simulation including models developed using SystemC. Finally, the VCI supports complete systems verification through the integration of system software modules with Verilog simulations.

## COMPLEX DATA TYPES

TestWizard supports a comprehensive set of complex data types that can be used to represent transactions from within a Verilog or VHDL model. Record datatypes provide the capability to represent user-defined structures. TestWizard supports functions to create and access records and its fields. Records can be nested to form hierarchical record types and linked lists. List datatypes provide the capability to store a collection of string, number, and number range constants. Lists are mutable and TestWizard supports functions to create, append, and query, and extract values. Lists can be hierarchical too.




**RECORD DEFINITION FOR STS-3c POS PROTOCOL**

```
$tb_vrecord("sts192_short_frame",
"a1",     u_int8_t,     /* framing          (SOH) */
"a2",     u_int8_t,     /* framing          (SOH) */
"c1",     u_int8_t,     /* sts id           (SOH) */
"b1",     u_int8_t,     /* bip-8            (SOH) */
"e1",     u_int8_t,     /* orderwire        (SOH) */
"f1",     u_int4_t,     /* user             (SOH) */
"d1",     u_int8_t,     /* data com         (SOH) */
"d2",     u_int8_t,     /* data com         (SOH) */
"d3",     u_int8_t,     /* data com         (SOH) */
"h1",     u_int8_t,     /* pointer          (LOH) */
"h2",     u_int8_t,     /* pointer          (LOH) */
"h3",     u_int8_t,     /* pointer action   (LOH) */
"b2",     u_int8_t,     /* bip-8            (LOH) */
"k1",     u_int8_t,     /* aps              (LOH) */
"k2",     u_int8_t,     /* aps              (LOH) */
"d4",     u_int8_t,     /* data com         (LOH) */
"d5",     u_int8_t,     /* data com         (LOH) */
"d6",     u_int8_t,     /* data com         (LOH) */
"d7",     u_int8_t,     /* data com         (LOH) */
"d8",     u_int8_t,     /* data com         (LOH) */
"d9",     u_int8_t,     /* data com         (LOH) */
"d10",    u_int8_t,     /* data com         (LOH) */
"d11",    u_int8_t,     /* data com         (LOH) */
"z1",     u_int8_t,     /* growth           (LOH) */
"z2",     u_int8_t,     /* growth-febe      (LOH) */
"e2",     u_int8_t,     /* orderwire        (LOH) */
"j1",     u_int8_t,     /* trace            (POH)*/
"b3",     u_int8_t,     /* bip-8            (POH) */
"c2",     u_int8_t,     /* signal label     (POH) */
"g1",     u_int8_t,     /* path status      (POH) */
"f2",     u_int8_t,     /* user channel     (POH) */
"h4",     u_int8_t,     /* indicator        (POH) */
"z3",     u_int8_t,     /* growth/DQDB      (POH) */
"z4",     u_int8_t,     /* growth           (POH) */
"z5",     u_int8_t,     /* growth           (POH) */

"payload",  hdlc1_t     /* POS payload (PAYLOAD) */
);

$tb_var_record("sts192_short_frame", sts192_1);
$tb_record_new(sts192_1);

$tb_save_record_field(8'hf6, sts192_1, "a1");
$tb_save_record_field(8'h28, sts192_1, "a2");
$tb_save_record_field(8'h01, sts192_1, "c1");
$tb_save_record_field(8'h00, sts192_1, "b1");
$tb_save_record_field(8'h00, sts192_1, "e1");
$tb_save_record_field(8'h00, sts192_1, "f1");
$tb_save_record_field(8'h00, sts192_1, "d1");
$tb_save_record_field(8'h00, sts192_1, "d2");
$tb_save_record_field(8'h00, sts192_1, "d3");

$tb_save_record_field(hdlc2, sts192_1, payload");
end
```

***TestWizard's integrated record data type supports modeling complex transactions, generating random record field values, streaming record data over a signal interface, and storing and querying records from a transaction database.***

With Complex Data Types, testbenches can be written using multi-dimensional arrays, field-named structures, or any grouped or nested combination of these types. These capabilities make it much easier to write tests and transactors for designs that interface to packet-oriented interfaces. For example, when verifying routers, switches, or other networking/telecom devices, Complex Data Types can be used to work with items such as IP packets, ATM cells, Sonet frames, Ethernet frames, or MPEG frames directly in a Verilog/VHDL test or transactor. Random testing is also supported as any record field value can be assigned based random test parameters.

## SELF CHECKING TESTS

In order to verify correct system operation involving tests that many employ 10s to 1000s of transactions, automated result checkers are required to confirm operation or report simulation errors. It is impossible to expect a human to inspect the results manually and find all possible design violations. Result checkers can be either static or dynamic. Static checkers rely on comparing actual results to a set of expected results generated in advance. This is similar to using golden test vectors to compare against. Dynamic checkers verify transaction results during the simulation either for each transaction or after a pre-determined sequence. TestWizard supports Transaction Databases and Record and Signal History functions to make it easier to implement self-checking tests by logging and tracking how the system processes transactions. Typically transaction logging requires transaction ID tagging and transaction completion notification to trigger result checking. When an error is detected, the transaction database can be interrogated or dumped to provide useful insights into the cause of the failure. However that is only part of the story. TestWizard also supports assertion checking and feature-level coverage analysis that further validates the design.

## TRANSACTION DATABASE

Transaction logging provides the capability to store transaction-level data in a database which can later be accessed as part of a result checker. TestWizard's transaction database facility allows one or more databases to be created that can be queried by index, record type, record-field value, and time-based searches. This provides a powerful query mechanism to sift through multiple transaction histories and extract just the right transactions to perform a particular result check. For communications designs which do not require strict transaction

completion ordering, transaction databases are a must have. Transaction databases are much more powerful than mailboxes and queues because transaction database entries can be of any transaction record type.

## ALGORITHMIC TEST GENERATION (DIRECTED OR RANDOM)

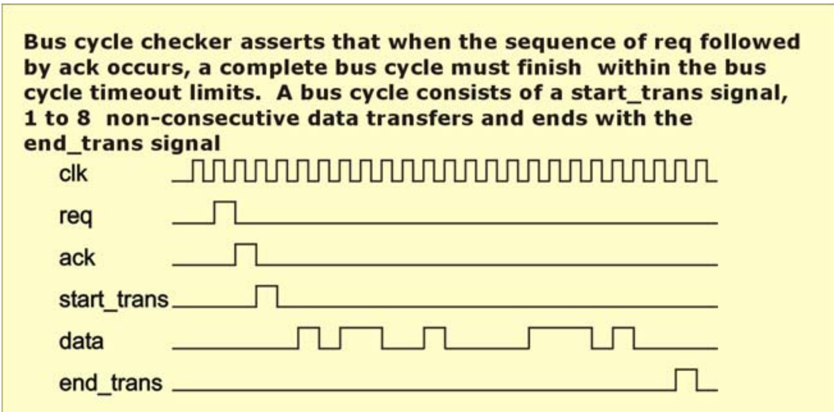
TestWizard supports algorithmic testbench generation. Designers can program transaction stream generators using Verilog or VHDL that generate the transaction sequences for each interface of the design being verified instead of having to manually create the long and detailed transaction sequences. This capability can easily be used to perform directed, exhaustive, or random testing of specific functions. TestWizard supports multiple constrained random case generators including weighted distribution, cyclic, boundary case, and multi-variable relations which makes generating any profile of transactions possible.

***Transaction databases support powerful store and query capabilities enabling complex end-to-end transaction result checking to be performed easily and effectively***

```
// Verify LDP Map Reply (Egress) by finding corresponding LDP Map Request
// on Ingress for IP src address.  LDP TCP Port = 646
$tb_xlog_find_value(top.inj0.xdb, xdb1, "th_dport", 646, 0);
xdb1_size = $tb_xlog_total(xdb1);
if (verbose)
   $display("IP LDP Map Requests Pending, size=%d", xdb1_size);
if (xdb1_size > 0)
  begin
  $tb_xlog_find_value(xdb1, xdb2, "src_addr", addr, 0);
  $tb_xlog_record_at(xdb2, 1, tmp_tcp3, time2);
  $display("\t\t%m MON, IP LDP Map Request Ingress, time=%d", time2);
  end
else
  begin
  $display("\t\t%m No IP LDP Map Requests Pending, Error, time=%d", time2);
  end
$tb_xlog_free(xdb1, xdb2);
```

## ASSERTION CHECKING

Temporal protocol checkers verify that system operational properties involving signal interface protocols and architectures are adhered to throughout a simulation run. Assertion checkers have several advantages including requiring no input stimulus, run "on the fly" throughout simulation, provides comprehensive coverage, and are reusable at module- and system-level as well as between projects. Using TestWizard's concise and powerful temporal function extensions allows designers to fully exploit comprehensive system and signal interface protocol validation.




TestWizard also supports Accellera's Sugar/PSL language. Design properties modeled using Sugar/PSL can be converted to TestWizard's Verilog and VHDL built-in assertion functions and execute within TestWizard's high performance and efficient assertion checking engine.

```
always @(negedge clk)
    if (data) -> xfer;

initial begin
  data_SEQ = $tb_seq_repeat($tb_range(1,8), xfer);
  bus_cycle_SEQ = $tb_seq($tb_posedge(clk), ack, start_trans,
                     data_SEQ, end_trans );
  #1 $tb_seq_trigger(bus_cycle_SEQ, bus_cycle_RESULT);
  end

always @(bus_cycle_RESULT) begin
  $tb_update_profile(bus_cycle_trigger_PH);
  if (bus_cycle[0] == 1) $display("MON@%0t, bus_cycle violation", $stime);
```

***TestWizard's assertion checker engine supports advanced temporal logic functions to model complex protocols and provides effective assertion violation handling***

## FEATURE-LEVEL COVERAGE ANALYSIS

Without being able to measure actual functional test coverage against the verification test plan, verification engineers are operating in an open-loop system. Test pass rates and bug rates provide only indirect measures of functional coverage. Functional coverage analysis provides a formalized method to determine how thoroughly a system's features have been exercised. TestWizard's Functional Coverage Profiling functions enable designers to develop coverage models that define how to monitor the system and how to calculate and report coverage measures. Functional Coverage Analysis is essential for random testing methods to understand how effectiveness the testing is. Specifically coverage analysis can weed out duplicate and ineffective testcases, identify areas that are not being tested, and identify key tests for check-in and regression tests.

TestWizard's functional coverage profiling operates "on the fly" during simulation and can provide adaptive control over overall run length of the simulation based on dynamically querying the coverage monitors. Incremental coverage analysis is supported to gather a complete picture of verification for an entire regression testsuite.

## ADAPTIVE TESTING

TestWizard can adapt test stimulus based upon the internal state of the simulation and its feature-level coverage monitors. This eases test development for systems that have indeterminate transaction processing characteristics and minimizes the effort to maintain testbenches as the design changes. Using Adaptive Testing, transaction sequences can be shaped at runtime to hone in on coverage requirements.

```
// Setup transaction packet verifier profiler
initial
  begin
  $tb_list_append_values(verify_packet_LH,
               "verify_ip_icmp_timeout_msg_latency",
               "verify_ip6_icmp_timeout_msg_latency",
               "verify_ip6_flow_latency",
               "verify_mpls_stack_forwarding",
               "verify_ldp_map_request");

  $tb_list_append_values(verify_packet_H,
               `verify_ip_icmp,
               `verify_ip6_icmp,
               `verify_ip6_flow,
               `verify_mpls,
               `verify_ldp);

  $tb_profile_create(verify_packet_PH, verify_packet, verify_packet_H, verify_packet_LH);
  end

/* Update transaction packet verifier profiler */
always @(update_profile)
  begin
  $tb_update_profile(top.mon0.verify_packet_PH, top.mon1.verify_packet_PH,
      top.mon2.verify_packet_PH, top.mon3.verify_packet_PH);

  test_coverage0 = 25 * $tb_profile_coverage(top.mon0.verify_packet_PH);
  test_coverage1 = 25 * $tb_profile_coverage(top.mon1.verify_packet_PH);
  test_coverage2 = 25 * $tb_profile_coverage(top.mon2.verify_packet_PH);
  test_coverage3 = 25 * $tb_profile_coverage(top.mon3.verify_packet_PH);
  test_coverage = test_coverage0 + test_coverage1 + test_coverage2 + test_coverage3;
  if (verbose) $display("test_coverage=%3.3f", test_coverage);
end
```

***TestWizard's coverage analysis supports assertion-based and functional coverage. Formal measures provide guidance on effectiveness and efficiency of verification testing. Run tests until all coverage monitors and assertions are triggered to ensure comprehensive verification levels.***

## PLATFORM SUPPORT

Solaris, Linux, HPUX, Windows

## SIMULATOR SUPPORT

| | | |
|---|---|---|
| **Cadence** | Verilog-XL | NC-SIM |
| **Synopsys** | VCS | |
| **Model Technology** | ModelSim | |

## LOCATIONS AND FACILITIES

**U.S. Headquarters:** 2 Atwood Lane, Andover, MA 01810, Tel: 978 689 7286, Fax: 978 258 5889
**International Field Office**: 5F, No. 95, Sec. 1, Chong-Qing S. Rd., Taipei, 100, Taiwan, Tel: 886-2-23817029

**Sales**

| | | |
|---|---|---|
| Avery Sales and Support | 978 689 7286 | |
| Saphirus | 408 656 1950 | (California) |
| DesignAVS (designavs.com) | 503 708 9351 | (Western US) |
| Facet EDA Solutions (faceteda.com) | 888 883 2238 | (New England, New York) |
| Technical System Integrators, Inc. (tsieda.com) | 407 339 4TSI | (Southeast), 512 335 8499 (Texas) |
| HiTech EDA (hitecheda.com) | 202 462 1516 | (Mid-Atlantic) |
| BlackForest EDA (blackforest-eda.de) | +49-2132-137485 | (Europe) |
| Kaviaz Corporation Limited (kaviaz.com) | +886 (3) 6668668 | (Taiwan) |

**WEBSITE:** http://www.avery-design.com

Trademarks/Copyright ©2004 Avery Design Systems, Inc. All Rights Reserved.