

ATE Open Architecture Initiative

Purpose

This document provides an overview of ATE architectural attributes sought in designing the ATE Open Architecture. It may be distributed freely to promote dialog between interested parties

Background

To make fundamental improvements in; cost, competitive capability, platform support, and performance scaling, component ATE needs to move from proprietary platform architectures to modular solutions that scale and encourage multiple supplier participation. With a modular architecture, scalability will continuously improve as upgrade and specialty solutions emerge. We expect to see the same level of variety, competition, reuse, and innovation that the VXI and PXI standards have promoted in non-VLSI ATE. Proposed architectures will require careful consideration of the variety of environments encountered and performance expectations of VLSI ATE. Applications will include virtual every VLSI component type and tester usage condition.

Initiative Support Attributes

Listed below are the desired design attributes and capabilities.

Flexibility

Flexibility in ATE allows a platform to be configured in a variety of ways without resorting to extensive re-fit or customization. As an example, with 1024 tester channels implemented as 16 test modules of 64 channels each, a variety of platform configurations should be possible:

- ?? Sixteen independent (encapsulated) test sites of 64 channels each
- ?? Four test sites of 256 channels each
- ?? Two 512-channel test sites or one 1024-channel single test site tester.

Modularity

Chassis slots should be completely generic and allow any conforming test module to be inserted. Physical location of a test module within the chassis may depend on the electrical performance considerations of distance to the device under test or it could be constrained by physical spacing requirements for multiple contactor arrays in parallel test applications.

Scalability

Scalability is in terms of the platform's ability to remain viable over time. Fundamental limitations in test module size, power, cooling, data transfer rates, etc, will be the ultimate constraints to long-term capability enhancements. A careful balance between opposing goals – cost, size, and performance – needs to be crafted to allow this base chassis to support test module capability improvements for 10+ years.

Module capability will be confined by limitations in technology integration, available power, thermal density, cooling capacity, and the structure of the specified interface standards. Confining the most rigid aspects of the standard to the module/chassis interface will enhance long-term scaling.

Encapsulation

The concept of modularity should extend beyond HW implementation and include SW support, applications, diagnostics, calibration, etc. Third parties must be able to supply the differentiating aspects of the modular solutions they supply without significant integration and compliance complexities. Module encapsulation supports this requirement.

Complex user application SW cannot be maintained via low-level HW drivers. Conversely, a rich user application layer will become very complex if its implementation is defined only at the system level. It would be difficult to comprehend the success of this initiative if the system controller is managing low-level HW bit read/write in the traditional method. Instead, module control will likely require on board processing of higher-level instructions (linked to SW API's) to/from the system controller. Alternatively, a typical memory mapped hardware architecture could be used, but the number and size of the registers written by the controller would need to be minimized such that the local module controller is now doing most of the hardware setup and configuration based on simple register "commands" written by the system controller

Efficiency - Performance

VLSI ATE needs to perform efficiently in the High-Volume Manufacturing (HVM) environment. Generally, test-related cost scales as a function of test time and directly relates to the total capital outlay in the number of platforms required as well, as support and labor expenditures. Very complex VLSI test application flows need to execute completely in a few seconds, and 50msec incremental test time improvements are routinely pursued.

Other performance considerations include:

- ?? test program changeover, setup, and load/init time for main program and test patterns
- ?? test results processing
- ?? diagnostic/calibration run-time and periodic maintenance interval
- ?? equipment uptime: reliability and availability
- ?? time required to diagnose and repair

Even in non-HVM applications, performance efficiency is important. For example, tester time during engineering development needs to be productive to meet the time-value needs of the operations performed. These environments tend to stress the setup time performance of VLSI ATE and are an important consideration in overall utilization.

Vision – Elements of the OAI

Test Modules

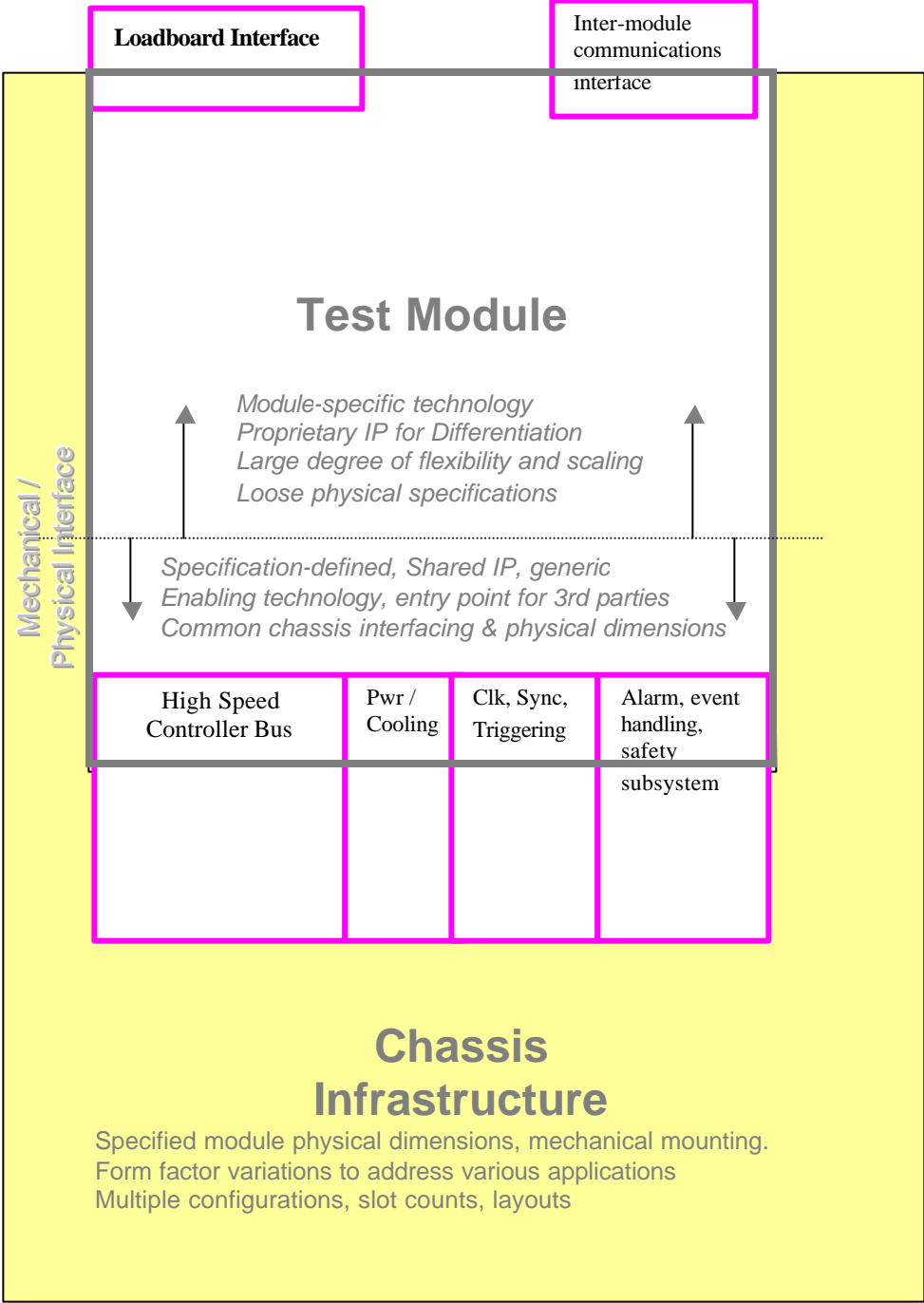
Test modules would contain everything required to generate stimulus and/or response test capabilities. They would communicate to/from the controller and other test modules via a high-speed bus interface and dedicated signaling lines. They would derive their power and cooling through standard connections to the chassis. Test modules would interface directly to the user's DUT fixture (loadboard) and, therefore, manage the performance of the electrical path as required for that module's capability. As performance requirements scale, the module's self-contained tooling interface will be appropriate for the cost and performance required.

Flexibility in the design and innovation of module capabilities needs to be maintained. At the same time, a well-defined (backplane) interface specification is required to allow multiple supplier participation. It is desired that interface standardization will be promoted via

technology enablers (interface ASIC's, simulation models, application support) to reduce the burden on new module designers. Module types will include device power supplies, digital tester capability (standalone pattern source/control, format and timing, and pin electronics), analog instruments, etc.

Module-specific flexibility should include provision for custom/proprietary inter-module communication connections. These connections would serve emerging performance challenges and promote technical differentiation between module offerings. The custom connection would be optional and confined only by physical location. The chassis would accommodate the space required but not support a specific electrical specification for this connection.

A key attribute of test modules is their ability to act as an independent tester site. Modules could be configured to act independently, or together with other modules as multiple encapsulated test sites, or as a single test site with larger channel count. Modules would be available to support a variety of performance, channel count, cost, etc ranges. Over time, improvements in device performance and integration would enable new test modules that continuously drive lower costs and better capability.

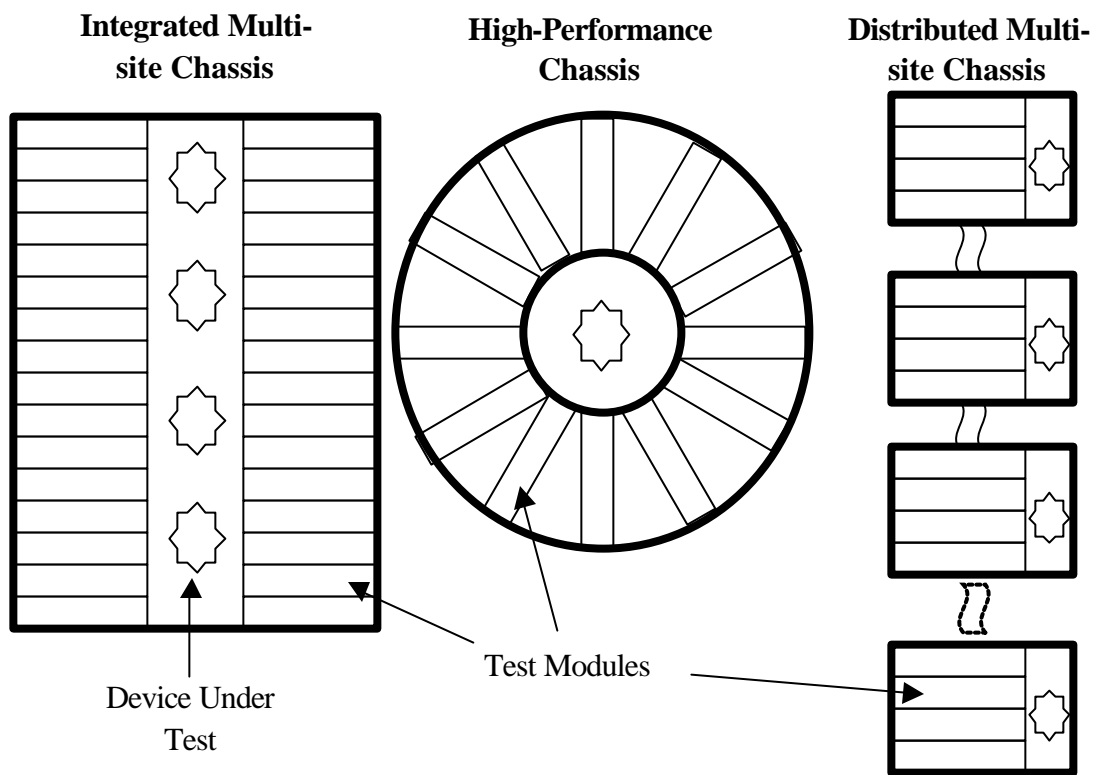


System Controller

The controller would perform a role similar to what is seen in today's ATE. Managing a complex user applications layer and high-speed real-time machine control requires a high performance CPU. Additional performance demands would occur as the controller manages multiple independent test sites and test module count and overall process independence increases. One approach may be the use of per-test site 'slave' controllers (**Site Controller**) as an optional way to improve multi-site performance when needed.

Chassis

The HW chassis supports physical test module mounting, power, cooling, mechanical alignment to material handling systems, safety and alarm monitoring, and the backplane/bus interfaces. It is expected that a variety of chassis choices from various suppliers would be available. Chassis configurations could be scaled for the particular application to accommodate size, form factor and cost considerations.



Flexibility in chassis requirements is encouraged. Traditional challenges such as mechanical docking to material handling equipment and configuring load boards with multiple contactors for parallel test can be addressed with innovative chassis form factors. A large, single-site chassis might be appropriate for high-speed, large pin count devices. Smaller sub-chassis configurations could be used for smaller pin count high volume parallel test applications.

Backplane Interface

The communications bus between the controller and test modules needs to allow for fast data transfer. Fast data transfer is needed to efficiently load test program content (digital pattern data for example), transfer large amounts of response data, and provide real time control and

message management. The high-speed controller bus should be structured to encourage ease-of-development from a variety of suppliers.

Additionally, some inter-module communication will be required for very fast signaling and coordination during test operations. Examples include test stop/start signals and connections for inter-module device triggering. In cases where high-speed or high accuracy inter-module communication exceeded the existing infrastructure, custom module links would be allowed. A system-wide clock would provide test module synchronization.

An error/event handling interface, independent of the main high-speed bus, would be required to manage safety monitoring and system power/thermal control.

Tooling Interface

The tooling interface requires specifications for dimensions, mechanical mounting, contactor placement, etc. It is expected that the actual test module to tooling connection will vary depending on the configuration of the installed test modules. Performance will dictate the connection type required and many factors will influence the density and layout of the loadboard interface. For this reason, the module will contain its own connection interface. A confining standard is not necessary or desirable for long-term scalability.

Compute Environment and Software Considerations

The controller and software need to support existing VLSI component test demands for test time performance, programming ease, test program usage models, etc. But to enable multiple developers to participate in an open architecture, the environment must be designed to allow capability extension over time without undue burden on developers or users.

Platform

First and foremost, it is encouraged that the solution be based on the Windows™ operating system and PC compute platforms. It is this OS and HW basis that best supports cost, modularity, scalability, and ease of development.

The system controller should be independent from the user workstation. The controller should be optimized for real-time control of the test modules. The controller (system/test head controller and/or the system workstation, depending on the system architecture) should be isolated from workstation overheads such as network I/O, keyboard, or graphic display. The goal is to achieve optimum consistent test time performance that does not vary with external conditions. The controller should implement multi-threaded real-time control processes. Therefore it should have a high enough bandwidth such that it can handle many test sites with ease. It will also require low (and more importantly) predictable and consistent latency. Additionally, it should allow for scaling to the particular application's performance demands. For example, if we assume the platform is not bus transaction limited, controller performance will establish the ceiling for multi test site control speed. Scaling in terms of faster CPU's, memory size, multiple CPU's etc. should not be limited by the controller architecture.

On the other hand, the system's local workstation acts as interface between the tester and the operator/network. This local workstation might connect to the controller with a point-to-point

fast Ethernet link or similar interface. It may be used to run the test cell's station controller software. On this platform, the operator manages the tester through GUI's and command line interfaces. The platform SW for interactive tools, datalog display, program loading, diagnostics, etc. will run on the local workstation. A PC/Windows™ based platform can leverage the large number of development tools available as well as low cost. The use of this architecture has side benefits in program and support portability (envision a field service engineer connecting his laptop which runs the tester OS to perform advanced diagnostics).

For offline development, a PC/Windows™ workstation should be all that is required to run the software necessary to create test programs, compile/link/load, and emulate tester operation. For large compute intensive operations such as test pattern compiles, system SW should be unbundled and capable of running independently to allow job batching across distributed platforms. SW tools for batch jobs should run on multiple platform types including UNIX, LINUX, and Windows™.

Performance

Program/pattern load and run times are critical. The architecture must support data transfer and real-time control at very high rates and requires that the bus data rate should not be a serious limitation. Additionally, pattern load time cannot be unduly burdened by the requirement to perform load time interpretations of items such as link table builds, label resolution, modulo address boundary restrictions, format definition linking, etc.

Run time is impacted by bus transaction rates and is heavily dependent on the level of interpretative versus deterministic items in the test flow. When a volume of device data needs to be analyzed in real time, the data transfer and controller time cannot slow down testing or have a negative impact on other test sites.

To support independent test sites comprised of numerous test modules, a centralized controller/bus architecture will be challenged. The overall architecture should minimize load time by allowing for broadcast pattern data and setup loads to multiple modules in different test sites. The system controller should support independent 'instances' of the test program for each test site in a multi-threading fashion.

In general, complex device test flows cannot tolerate execution 'stalls' as a parallel test site control strategy. Sites must act independently and with the same performance (total test time) in single and multi-site configurations. Modules should encapsulate sufficient intelligence to manage high-level instructions and not rely on traditional memory-mapped h/w register read/write from the central controller. The architecture should not preclude the use of per-test site controllers in multi-site applications that would overwhelm the bus and central controller.

Station control

The tester must be accessible remotely. A network interface to a remote workstation is common today. With the emergence of requirements such as e-diagnostics, it will be useful to establish an internet-based remote access solution. If a web browser interface is used, the remote workstation control can be architecture independent.

In general, the attributes of a SECS/GEM/TSEM interface should be supported to communicate with the customer station controller process. The station controller may run on the local workstation, or remotely via TCP/IP.

Multi-site support

A single test program should be instantiated for each site and not require any customization for parallel test. Separate from the test program, user management of the system-level parallel test strategy is needed. For example, in a 16-site test cell, a site that fails continuity tests on 4 devices in a row may be shut down and left unused until the lot completes – or, perhaps, until the next monthly PM.

Modularity

Contention, performance degradation and other integration problems need to be managed with modularity solutions that extend to platform SW. Controller communication with test modules should be restricted to fewer, small commands, not complicated hardware setup such as completely setting up a digital board state via a large number of direct register writes. This frees up the central controller and reduces bus traffic (but puts additional computing burden on the test module itself).

A DLL registration type approach will allow the module-specific API extensions to become accessible to the user. An API specification would include rigid requirements for some elements such as error handling, standard debug datalog, output record formats, regression testing, etc. The API specification would be very flexible in module control and read back operations.

Test program language

The test programming language is expected to be a combination of

1. A standard object-oriented language (i.e., C++) and
2. STIL-based test program “object” declaration language.

The STIL language syntax and object hierarchy should be followed and extended where STIL is incomplete (i.e., PMU setups and pattern generator scramblers). Tester OS APIs must provide read/modify/create/delete access to the STIL-based test program objects. Test module specific APIs must provide for asserting test program objects to the hardware (i.e., set up to run a list of test patterns, or assert a levels object), and more discrete manipulation of the test module features (i.e., modify a signal’s global mask state, or read or write pattern memory bits)

Test module documentation standards must be set that cover test programming objects and APIs, diagnostics modes and user switches. All documentation should be bundled and available within the development and usage environments and leverage techniques such as context sensitivity and hyper linked text. Internet-based documentation should be considered for timely updates and improving distribution.

Test programs need to be controlled and customized external to the code they are written in. Typically, data specific to the environment and test conditions needs to be communicated in a controlled fashion. Global variable constructs that are available to the test program and externally accessible are a common method to achieve this. A broad range of global types and with various interpretation levels (load time, init time, run time, etc) should be supported.

Test module drivers

A defined standard needs to specify the base set of software to be supplied with every test module, including

- 1) registration and initialization

- 2) test programming objects and APIs
- 3) calibration and diagnostics extensions
- 4) OS GUI extensions.

It is expected that when the tester operating system is brought up, some sequence of test module and associated driver registration must take place. At the beginning of the test program load process, a check must be performed to ensure that the necessary tester hardware is installed in the required module slots, properly configured and grossly functional. Each test module is anticipated to have its own unique set of STIL-like test program objects types and APIs, which extend the test programming language via header or include files. Each module type should provide service routines that extend the baseline calibration and diagnostics for the system, with all the appropriate flexibility.

It is expected that substantial effort will be invested in designing baseline standards for test program object and API standards, including provisions for extending the initial set of objects and APIs to support new module types over time. Extending the existing objects is seen as a method of reducing the proliferation of object types that vary only slightly.

Static objects (i.e., levels objects, PMU setups, pattern generator setups) might be stored in the test module's local controller so as to optimize test time and limit the number and complexity of transactions asserted by the system controller.

GUI extensions need to be carefully considered. Some generic interactive tools like shmoo plotting might simply have new axis objects available from the specific modules populated in the system. Other tools might be bundled with and wholly dedicated to work only with a specific module type.

Test Pattern handling (specific to digital device testing)

Digital device test patterns need to be established separately from the test program and not be 'compiled in'. Complex digital devices require 1000's of test patterns that are specified and bundled in pattern lists. These pattern lists contain lists of patterns, pattern control information, and other pattern lists. The user should be able to change the contents of the lists (as well as the test patterns themselves) without modification or re-compilation of the test program.

The test program references the named pattern lists and every tool and API should be able to refer to pattern lists and individual patterns as hierarchical objects. At test program initialization-time, selected lists and patterns designated within the lists are loaded according to user-defined search paths (again independent from the compiled test program).

The patterns are loaded in the tester's vector memory as efficiently as possible. Test pattern language constructs must be provided for definition of any digital pattern generator indirection memory (i.e., a scrambler memory) if there is a substantial load time, resource savings and/or test time advantage to be had by offline calculating bundled pattern sets and indirection memory contents.

The tester HW should support execution in any pattern sequence order independent of the load address without a break in the device test.

Error handling

High volume manufacturing is one of the applications that requires sophisticated detection and control of excursion events. 'Errors' are the class of excursions that do not necessarily force a stop to testing, and do not present a person or machine safety hazard. The architecture must incorporate a robust design for runtime error handling. This would include a standard set of error priorities and hardware-based default responses that will scale from site-specific to the full system level. User-programmable response to error conditions is highly desirable. Specifications are required for main controller notification and event logging. Response actions should accommodate the range from routine (anticipated) excursions to those that are fatal to continued test site execution (unexpected events).

The controller should manage an independent error handling thread that responds to system errors and responds with the appropriate action. There should be a class of errors for which user programmable responses can replace the default. For example, a DUT power supply can generally report an error for an over current condition. The user may want to stop the test program immediately and notify the station controller as it indicates a faulty loadboard, or it may be part of their test methodology to use an over current as part of a measure of the quality of the DUT and proceed with the appropriate test flow branch. Users should be able to write the response code for each particular event, as well as set the priority level for the event so that they can be processed in the desired order.

There should also be a class of errors that the response may not need to be user programmable. For example, there might be an error reported for a bulk power supply in the chassis that is out of voltage spec (or even powered down). This could be a case where the response would be standard such as: Print out the error message and disable (site) testing until the problem is fixed.

Alarm handling

In general, alarms alert and respond to excursion events that are not recoverable without direct intervention, can cause system damage, or present a safety hazard. There can be multiple alarms priority levels, the most important of which is the independent system/module EMO loop. This control function has the capability to shut down power to the system immediately. EMO switches at the tester system level can trip this loop when a user pushes them, as well as smoke detection, thermal overload, power faults, coolant leaks, etc. EMO loop capability must also be provided at the board level. Each board must be able to shut down the system in the same manner. The EMO loop needs to allow for expansion and accommodate new detection conditions requiring shut down. This control loop must be robust (HW-based) and have an independent power circuit. When it is tripped it powers down all of the other system power immediately. The system design should allow for EMO shut down without damage to components due to its uncontrolled nature.

The next level of alarms is less immediate in nature, and can be under SW or FW control, but handled by a separate controller that is completely independent of the module controller or tester workstation. This alarm level is used to monitor system parameters such as voltages, temperatures, currents, etc. If these parameters fall out of the allowable range then the controller may notify the system controller, or take some other appropriate action, one of which may be to do a controlled (complete or partial) power down of the system.

It is desired that system detection and monitoring facilitate maintenance activities. For example, bulk power supply voltage monitoring would be accurate enough to derive adherence

to the their calibration limits. This information would be used to automate and enhance maintenance procedures as well as provide real-time system health assessment.

Command line interface

The command line interface should use the same API commands as are used within a test program, but would be interpreted real time to avoid compilation. If the standard programming environment is already simple, interpretive and flexible in nature, then a separate command line interface may not be necessary.

The command line interface should allow calling script files of the commands to facilitate running canned routines for debug or characterization purposes. Again, depending on the base architecture of the programming environment this may not be necessary.