# TEST PROGRAMMING ENVIRONMENT IN A MODULAR, OPEN ARCHITECTURE TEST SYSTEM

*Ankan Pramanick, Ramachandran Krishnaswamy, Mark Elston, Toshiaki Adachi, Harsanjeet Singh and Bruce Parnas*

Advantest America R&D Center Inc.,
3201 Scott Blvd., Santa Clara, CA 95054, USA

*Leon Chen*

CPU Tech
5731 W. Las Positas Blvd., Pleasanton, CA 94588, USA

## Abstract

*This paper addresses two key concepts in device test program development*: *test class programming and pattern management. These are explored in the context of an open architecture test system, where the primary requirement is the flexibility to integrate externally developed capabilities into the system. Development against an open architecture test system includes the integration of software-based solutions* (*such as user-developed test classes*) *and third party hardware modules, including the software necessary to support the modules. This paper focuses on the open architecture facets of test programming and pattern management, as embodied in the* OPENSTAR™[1] *specification. The software for Advantest Corporation's* T2000 *system is used as a concrete example for highlighting these concepts.*

## 1 Introduction

Several innovative approaches to cutting the cost of testing have been presented over the last decade or so [1] — [5], ranging from parallel *device under test* (DUT) operation [1], which is now a standard feature of most high-end testers, to a call for an *open architecture* (OA) test system [3]. It is in this last area that there has recently been a lot of interest [6] — [13], both in the semiconductor manufacturing community, and the ATE vendor space. The OPENSTAR™ test system [13], proposed by the *Semiconductor Test Consortium* (STC), is such a system, whose goal is to provide an architectural solution, under which third party software and instruments can be developed, certified individually, and integrated reliably into ATE. This architecture is aimed at allowing deployment of ATE by a customer in a "plug and play" manner.

[1] OPENSTAR™ is a trademark of the *Semiconductor Test Consortium* (STC).

The essential hardware attributes of an open architecture ATE have been discussed in [6], [10], [13]. To make fundamental improvements in cost, competitive capability, platform support, and performance scaling, it is necessary that ATE move from proprietary platform architectures to modular solutions that scale and encourage multiple supplier participation [6].

As mentioned in [8], a key component in enabling such an architecture is the system software, and its ability to support extensions with minimal required changes. Ideally, there should be no need for re-compilation into executable machine code for additional or new vendor-provided functionality.

For the end user, two very desirable elements in a test system are a *test control programming* environment, and a software runtime *measurement* or *test* object. Such a system should provide

1. a *modular* development environment, permitting users to write individual components dealing with different aspects of the test, and then mixing and matching them in various ways;
2. an *iterative* development environment, with opportunities for minimizing recompilation efforts when only parts of a test program, and not the entire test program, change;
3. a way to *separate* test methodology from DUT-specific data, thereby allowing code reuse in their test objects;
4. a way to *generalize* test methodology providing objects; and
5. a degree of *insulation* from vendor-dependent characteristics of the OA test system.

Equally important to the testing process is the management of the test patterns used to exercise and evaluate the DUT. While the *Standard Test Interface Language* (STIL) [14] is


0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

especially suited for representation of digital device patterns, in the context of an open architecture test system, the pattern management scheme must allow for seamless inclusion of user-defined pattern formats, even some that are not intended for digital device testing. The pattern management system must allow the integration of these user-defined pattern formats in such a way that the specific contents of these formats may remain unknown to the framework; only the means to access these data are relevant.

Traditionally, a high-level language such as C, C++ or Java, or a proprietary language from a particular vendor, has been used for test programming. Currently, there are testers from several vendors with disparate and incompatible programming environments and proprietary data manipulation methods for DUT test implementation. These often require a major effort in coding and making new releases to accommodate new functionality. Recently, the approach of [12] is one of the first that targets an OA paradigm, and deals with some of the above goals.

In contrast to currently existing test systems, the OPENSTAR™ standard proposes a complete test programming environment in support of all the above goals [13]. While using the ANSI/ISO C++ language for test programming, it, at the same time, allows hiding of the complexity of that language from the less sophisticated user through a much simpler test programming language, the *OPENSTAR™ Test Programming Language* (OTPL) [15], and a compiler to transform programs written in that language to C++. The standard also provides for a pattern management framework designed to allow the incorporation of externally-defined pattern syntax in support of third party vendor modules.

It must be clearly understood here that OTPL is neither necessary nor recommended for test programming in the OPENSTAR™ system, which can be accomplished perfectly well through only the use of ANSI C++; the intention was not to create "yet another proprietary language", but to provide a simpler programming tool for users who did not want to be bothered with using C++ directly.

This paper uses the *T2000 System Software* (TSS) — incorporated in Advantest Corporation's T2000 next generation open architecture tester — as a concrete example of an implementation of the OPENSTAR™ specifications. The user should first refer to a companion paper in these proceedings [16] for a discussion of the overall software architecture of the OPENSTAR™ system, which is essential to the proper understanding of the material presented in this paper, and which, for the sake of brevity, is not repeated here.

The rest of this paper is organized as follows. Section 2 provides a short overview of the software programming environment in the TSS. Section 3 provides a brief description of how the TSS test development environment supports modularity, and an iterative development environment for user test programs. Sections 4 and 5 introduce and describe the techniques used in the TSS to realize the goal of generalizing a user measurement (test) object, while section 6 describes pattern management in an open architecture framework. Finally, section 7 provides a summary. Note that a discussion of providing insulation from vendor-specific elements and enhancements in an open architecture environment, being an entire topic in itself, is beyond the scope of this paper; it is discussed in detail in [13].

# 2 The Programming Environment

The native operating system platform for the TSS is Microsoft Windows, chosen in order to leverage the large number of development tools available, as well as for its low cost. This also has benefits in program and support portability (e.g., a field service engineer could connect his laptop which runs the tester operating system to perform advanced diagnostics). However, for large compute-intensive operations (such as test pattern compiles), the relevant software is unbundled, and capable of running independently to allow job scheduling across distributed platforms.

## 2.1 Native Interface Language

ANSI/ISO standard C++ is the native interface language of the OPENSTAR™ system, and hence, of the TSS platform. There are *performance* and *platform* issues associated with the choice of any approach for providing a system interface. C++ was chosen as the best candidate for addressing both these concerns. Other popular system interface models, such as Microsoft's *Component Object Model* (COM), have an adverse impact on both these fronts as far as OPENSTAR™ requirements are concerned, and yet other choices such as the *Application Binary Interface* (ABI) on Sun Microsystems' Solaris OS are obviously not suited for the Windows platform. Another popular language with abstract interface support, Java, though very suitable for components such as graphical user interfaces (GUIs), is demonstrably slower than C++ when it comes to systems programming. Since it was judged that a source-level compatibility, in terms of a widely-applicable and efficiently implemented language, was essential, C++ was the best choice. Of course, there are a multitude of options available (to provide a layer over the OPENSTAR™ C++ interfaces) that allows a third party to integrate into any OPENSTAR™ system with a language of their choice, but this is a responsibility of the third party.

# 3 Test Programming in TSS

The principal user component of the TSS programming environment is the *test plan*. A *test plan* is a test program written by the test engineer. The test plan uses *measure-*

*ment* or *test classes*, which realize the separation of test data and code (and hence, the reusability of code) for particular types of tests.

The test plan may be written directly as a C++ test program, or described in a set of *test plan description files*, using the OPENSTAR™ Test Programming Language (OTPL) [13], [15]. These files are processed by the OTPL compiler to produce C++ code. The generated C++ code is then compiled into the executable test program, which is in the form of a *dynamic link library* (DLL). The data required for populating a test class instance are parameterized by the user in the test plan description files.

## 3.1 OTPL Support for Modularity

The OTPL defines the syntax and semantics of files that provide the input for a test program. One of the objectives to be met in the design of this language was *modularity*. A test programming language supports *modular development* if it permits users to write individual components dealing with different aspects of the test, and then permits these components to be mixed and matched in various ways to yield a complete test program. To this end, the OTPL allows for the information needed for a test program to be assembled together from several files. These files are in support of several OTPL *sub-languages*, which include, among many, a *user-variables* sub-language, a *levels* sub-language, a *timings* sub-language, a *bin definition* sub-language, a *test plan* (flow) sub-language, a *pre-headers* sub-language (for test classes), and so on.

A single test program comprises a single *test plan* file, and the files it *imports*. An "import" refers to another file with data that is either directly referenced by the *importer* (the file that specifies the import), or is imported by some other file directly referenced by the importer. The test plan file could define OTPL objects within it, or it could import this information from other files. The OTPL allows any of the above language components to be either in their own individual files, or directly in-lined into a test plan file. Note that the test plan is similar in concept to a C/C++-language `main()` function.

## 3.2 Building an OTPL Test Program

The OTPL test program description files specify the objects used in a user *test plan*, and their relationships to one another. These files are translated to C++ code by the OTPL compiler, and are meant to be executed on a T2000 *Site Controller* (cf. [16]) in the form of an implementation of the standard OPENSTAR™ interface `ITestPlan`. This code is packaged into a Windows *dynamic link library* (DLL), which can be loaded onto a Site Controller. This DLL is known as the *test program*. The test program DLL is generated to have standard known entry points that the Site Controller software can use to generate and return the `TestPlan` object it contains.

The process of conversion from a test plan OTPL description to an implementation of `ITestPlan` is accomplished by the OTPL compiler in two phases: *translation* and *compilation*, as follows:

- In the translation phase, the OTPL compiler processes a test plan file (and the various other OTPL files it imports), as well as the OTPL *pre-headers* for all the test types used in the test plan (cf. §5.2). In this phase, it creates the C++ code for the `TestPlan` object and the C++ headers for the test types encountered, along with all other supporting files, such as project makefiles, DLL "boilerplate" code, etc.
- In the compilation phase, which occurs after the OTPL compiler has created the necessary files, a native C++ compiler is invoked to compile the files and link them into a DLL.

### 3.2.1 Iterative Development Support

Consider the conversion and compilation of a test plan description file `MyTestPlan.tpl` that uses (i.e., imports) a levels definition OTPL file `MyLevels.lvl`. If, after building the `MyTestPlan` DLL, the user made a change to the definitions of *levels* (i.e., operating parameters) in `MyLevels.lvl`, he would then invoke the OTPL compiler again, passing it the main test plan description file `MyTestPlan.tpl`. The OTPL compiler would recognize that the main test plan file is unchanged, so that `MyTestPlan.h/.cpp` would not be recreated. However, while processing the main test plan file, it would see that the `MyLevels.lvl` file has changed. Therefore, it would recreate the `MyLevels.cpp` file. The native C++ compiler would then be invoked to rebuild the test plan DLL. This avoids recompiling `MyTestPlan.cpp`, and only compiles `MyLevels.cpp` and re-links the DLL.

Thus, the modular, separate file-based approach in OTPL programming, together with the OTPL compiler support for regenerating C++ output files only if the generated source is different from the present source, is especially useful in cutting down re-compile and re-link times for large test plans that take a significant amount of time to compile.

## 3.3 Running a Test Program

The Site Controller software loads the test program DLL — and test class DLLs — into its process space and calls a "factory" function [17] within the DLL to create an instance of the `TestPlan` object. The `TestPlan` object in turn uses "factory" functions to create test class instances for the test types it contains from the corresponding test class DLLs. Once the `TestPlan` object and its subordinate test class objects have been created, the Site Controller

software can then execute the test plan or interact with it in any other way.

## 4 Test Objects in TSS

If a test program is written for a DUT, in general more than ninety percent of the program code is *data* for device test, and the rest is the "real" code which realizes the test methodology. The data is DUT-dependent (e.g., power supply conditions, signal voltage conditions, timing conditions, etc.). The test code consists of methods to load the specified device conditions on to ATE hardware, and also those needed to realize the user specified objectives (such datalogging, etc). The TSS framework provides a hardware-independent test and tester object model that allows the user to perform the task of DUT test programming.

Since test methodology is a significant component in device test quality and tester productivity, a limited number of persons are generally permitted to create and modify test programs, which should thus be encapsulated and hidden from the end users. This separation results in the safer operation of the system. Moreover, to increase the reusability of test code, such code should be independent of any device-specific data (e.g., pin name, stimulus data, etc.), or device-test-specific data (e.g., conditions for DC tests, measurement pins, number of target pins, name of pattern file, addresses of pattern programs, etc.). If code for a test is compiled with data of these types, the reusability of the test code would decrease. Therefore, any device-specific data or device-test-specific data should be made available to the test code *externally*, as inputs during code execution time.

In the TSS, a *test class*, which is an implementation of the standard OPENSTAR™ `ITest` interface, realizes the separation of test data and code (and hence, the reusability of code) for a particular type of test. Such a test class could be regarded as a "template" for separate instances of it, which differ from each other only on the basis of device-specific and/or device-test-specific *data*. Instances of test classes are specified in the test plan file.

Each test class typically implements a specific type of device test or setup for device test. For example, *functional*, *AC* and *DC parametric* tests could be implemented by separate test classes. While the system might implicitly know about the interfaces for certain basic and often-used test classes (for which the system itself provides implementations), custom test classes can also be freely used in test plans. The use of these custom test classes *do not require* a recompilation of system code that deals with the classes, and the making of any new release. This is because OTPL supports a *generalization* mechanism whereby all test classes are required to derive from the standard OPENSTAR™ `ITest` interface, and that is all the system deals with in its interaction with any test object. Customization is made possible through the capabilities in OTPL to describe a custom test (with its own parameters) as a *specialization* of the `ITest` interface.

Test classes allow the user to configure class behavior by providing parameters that are used to specify the options for a particular instance of that test. For example, a functional test may take parameters called `PList` and `TestConditions`, to specify the list of patterns to execute, and the *levels* and *timing* conditions for the test, respectively. Specifying different values for these parameters (through the use of different "*Test*" blocks in the test plan description file) allows the user to create different *instances* of a Functional Test. Figure 1 shows how different test instances would be arrived at from a single test class. These instances would be specified in the test plan, and in general, can obtain their specific data either from definitions of test parameter objects in the test plan, or algorithmically at runtime, or even interactively at runtime through applications that are able to manipulate their interfaces.

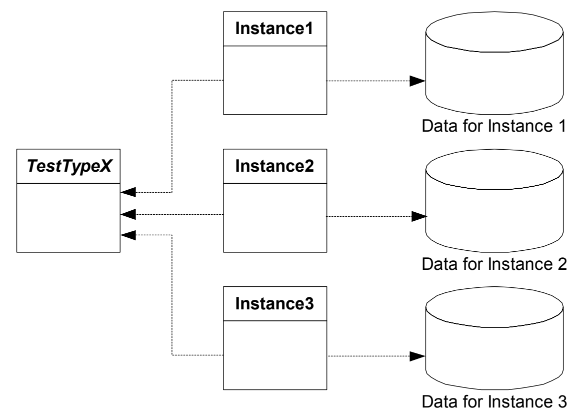


**Figure 1. Test Class Use in TSS**

As mentioned above, test classes derive from `ITest`. That is, these are C++ classes that implement the standard OPENSTAR™ `ITest` interface. In addition to the methods specified for the `ITest` interface, these classes provide the test-specific intelligence and logic required to perform specific classes of device test. Test classes also implement the standard OPENSTAR™ `IFlowable` interface. As a consequence of this, instances of test classes can be used in *test flows* — i.e., as *flow items* — in the test plan, to create a complex execution sequence of device tests.

From the user's perspective, these classes should be designed to allow the OTPL compiler to take the description of the tests and their parameters from the test plan file and generate correct C++ code, which can be compiled and linked to generate the test program. The next section shows how generalization of test class descriptions, allowing flexible customization of test classes, is made possible in OTPL.

## 5 Generalization of Test Classes

OTPL provides a generic mechanism for defining any test class in the system. Thus, test class creators can use this mechanism to create custom test classes that can be loaded and used by the system without any additional facilities being necessary. Test class creators can develop their own classes implementing the `ITest` standard interface. While it is possible for them to do so entirely in native C++, OTPL provides a mechanism that allows the customization of user test classes in a much simpler manner. This section presents the *pre-header sub-language* of OTPL that enables this. First, we discuss the requirements for an *introspection capability* on the part of the system to make this possible.

### 5.1 Introspection Capability

If an object of a test class could be interrogated regarding its methods and signatures, then it could be verified that the appropriate parameters are available for inclusion in the generated source code. Such a feature would be very useful for error checking and validation during the translation phase. If the test engineer made a mistake in the names of parameters, or the number (or possibly the types) of arguments to these parameters, the translation phase could catch it and provide a meaningful error message at translation time instead of waiting for a compile-time error message from the C++ compiler. This would be more useful, and probably more understandable, to the test engineer.

*Introspection* refers to the ability to ask an object to look within itself and return information regarding its attributes and methods. Some languages, such as Java, provide this ability as a part of the language. Other languages, such as Visual Basic, impose such a requirement on objects intended to be used with it. C++ makes no provisions for this feature.

This method also lends well to providing for default parameter values, as well as indications of optional parameters. In addition, if this capability is provided as a part of the implementation of all test classes, then GUI applications could also use this information to dynamically build up dialogs and other user interface elements to help engineers make effective use of these classes.

However, the disadvantage of providing this type of a feature is that it adds a level of complexity to the implementation of both the OTPL compiler and the test classes. This is due to the fact, as mentioned above, that C++ does not provide language support for this feature[2].

[2] As we have seen in §2.1, languages that do have native support for introspection were not deemed efficient enough for the OPENSTAR™ requirements.

These complexities are offset in the TSS through a mechanism that provides, in lieu of full introspection, a method that allows the test class developer to fully specify, in a *single* text-based source file (per test class), the public methods/attributes of the test class that the developer has designated as the ones required to parameterize the class.

It is important to emphasize the need for a *single* source: one would not want to have the description of the parameterization interface of a test class in one file, and the C++ interface description in another *independent* (header) file, and then be burdened with the need to keep both sources synchronized. Towards this end, the "text-based" description is *embedded* in a *pre-header* file for the test class. This is used by the OTPL compiler for limited introspection, and for generating the C++ header for the test class. The generated C++ header file is the one used to finally compile the test class C++ code.

### 5.2 OTPL Pre-Headers Sub-Language

Consider the following excerpt from an OTPL test plan source file, which defines an instance `FunctionalTest1` of a test of type `MyFunctionalTest` (the OTPL language keywords have been emboldened, and their precise meanings are not relevant in this discussion, except to note that a **`TestCondition`** is used to define a selector-resolved instance of a *condition parameter* for a test, such as *levels* or *timings*):

```
...
TestCondition TC1
{
   # A previously defined group for levels
   TestConditionGroup = TCG1;
   Selector = min;
}

TestCondition TC2
{
   # A previously defined group for timings
   TestConditionGroup = TCG2;
   Selector = min;
}
...
Test MyFunctionalTest FunctionalTest1
{
   # Previously defined pattern list
   PListParam = patList1;
   TestConditionParam = TC1;
   TestConditionParam = TC2;
}
```

**Listing 1. Test Plan OTPL Code Example**

Assume that `MyFunctionalTest` is implemented as a C++ class having base classes `Test1` and `Test2`, and having members which are a `PList`, and an array of `TestConditions`. The OTPL compiler needs to know about the types of the members of `MyFunctionalTest` in order to recognize that the above declaration of `Functional-`

Test1 is legal. Furthermore, in order to generate a C++ object declaration for FunctionalTest1, a C++ header for the class MyFunctionalTest needs to be constructed. This requires OTPL to also know about the base classes, if any, of the MyFunctionalTest class, the names of its members, and other such information.

Rather than building in the knowledge of a MyFunctionalTest into OTPL, the definition of what a MyFunctionalTest entails is specified through the *pre-header sub-language* of OTPL. This forms the basis of generalizing the test class description process. The pre-header sub-language provides the OTPL compiler with the information it needs to both recognize the legality of OTPL declarations, and to generate C++ headers and object declarations that correspond to an OTPL declaration.

One could thus write a pre-header, MyFunctionalTest.ph, that supports the above parameterization as follows (assume that pre-headers are available for the base test classes Test1 and Test2; note that the meanings of OTPL keywords, such as **TestClass**, and OTPL compiler variables, such as **$ClassName**, will become clear as one progresses through the example exposition):

```
Import Test1.ph;  # For base class Test1
Import Test2.ph;  # For base class Test2

TestClass = MyFunctionalTest; # The name
PublicBases = Test1, Test2; # Base test classes

# The parameters list:
Parameters
{
   #
   # The following declaration specifies that
   # a MyFunctionalTest has
   #  - one parameter of OTPL type PList,
   #     stored in a member named m_pPatList;
   #  - a function to set it, named
   #     setPatternTree; and
   #  - a textual description for it.
   #
   PList PListParam
   {
      Cardinality = 1;
      Attribute = m_pPatList;
      SetFunction = setPatternTree;
      Description = "The PList parameter "
                    "for MyFunctionalTest";
   }

   #
   # The following declaration specifies that a
   # MyFunctionalTest has
   #  - one or more parameters of OTPL type
   #     TestCondition,
   #  - stored in a member named
   #     m_testCondnsArray;
   #  - a function to set it, named
   #     addTestCondition; and
   #  - a textual description for these.
   #
   TestCondition TestConditionParam
   {
      Cardinality = 1-n;
      Attribute = m_testCondnsArray;
      SetFunction = addTestCondition;
      Description = "The TC parameter for"
                    "MyFunctionalTest";
   }
}

#
# The section below is part of the pre-header
# which is an escape into C++ code.
#
# Everything below is reproduced verbatim
# in the generated header file, except the
# compiler variables prefixed with $s.
#
CPlusPlusBegin

$ImportDeclarations

namespace OFC
{
class $ClassName : $PublicBases
{
// Array types for parameters storage:
$ParamArrayTypes

public:
   $Enumerations

   virtual void preExec();  # ITest method
   virtual void exec();     # ITest method
   virtual void postExec(); # ITest method

   $ParamFunctionDeclarations
   ...

private:
   double m_someVar;

   $ParamAttributes
   ...
};

...
$ParamFunctionImplementations
...
} // End namespace OFC
CPlusPlusEnd
```

**Listing 2. OTPL Test Class Pre-Header Example**

The above will result in the generation of a C++ header file with a declaration for the MyFunctionalTest class (implementing the ITest interface) as shown below.

### 5.2.1 C++ for Parameterized Test Classes

As the OTPL compiler processes the above pre-header file, it builds up the values of the OTPL compiler variables such as **$ImportDeclarations** (for Test1 and Test2), **$ClassName** (i.e., MyFunctionalTest), **$ParamArrayTypes** (a C++ typedef necessitated by the declaration of "**Cardinality**=1-n" for the TestConditionParam parameter), and others.

This enables it to then create the following C++ header by reproducing the C++ code (between **CPlusPlusBegin** and **CPlusPlusEnd**) verbatim, while expanding in the *values* (shown in *italics*) of the OTPL compiler variables **$ImportDeclarations**, **$ClassName**, etc., at the indicated places. From MyFunctionalTest.ph, it thus creates the following C++ header file MyFunctionalTest.h for the MyFunctionalTest test class:

```
#include <ITest.h>   # System header
#include <Test1.h>   # $ImportDeclarations
#include <Test2.h>   # $ImportDeclarations
#include <vector>    # C++ STL header
#include <Levels.h>          # System header
#include <TestCondnGrp.h>    # System header
...

namespace OFC
{
class MyFunctionalTest :    public ITest,
                            public Test1,
                            public Test2
{
// Array types for parameters storage:
public:
   typedef std::vector<OFC::TestCondition *>
         TestConditionPtrsAry_t;

public:
   virtual void preExec();  # ITest method
   virtual void exec();     # ITest method
   virtual void postExec(); # ITest method

public:
   void setPatternTree(OFC::PatternTree *);
   void addTestCondition(OFC::TestCondition *);
   ...

private:
   double m_someVar;

private:
   OFC::PatternTree *m_pPatList;
   TestConditionPtrsAry_t m_testCondnsArray;
   ...

};
...

inline OFC::String MyFunctionalTest::
      getXMLDescription()
{
   ...
}
...

} // End namespace OFC
```

**Listing 3. Generated C++ Code for Listing 2.**

The meanings of the keywords and the resulting constructs will become clearer when one reads §5.3 below. Note that the OTPL compiler would also generate the following code (in the test plan C++ source) for the MyFunctionalTest construct shown in Listing 1:

```
MyFunctionalTest FunctionalTest1;

FunctionalTest1.setName("FunctionalTest1");
FunctionalTest1.setPatternTree(&patList1);
FunctionalTest1.addTestCondition(&TC1);
FunctionalTest1.addTestCondition(&TC2);
```

## 5.3 Pre-Header Contents

A test class pre-header has, among others, the following declarations:

- *Optional* **Import**s, as in other OTPL files.
- The *mandatory* name of the test class.
- The *optional* list of public base classes.
- The *optional* **Parameters** section specifying the parameters of this test class (this is described in greater detail in §5.3.1 below).
- The *mandatory* C++ section providing a *template* for the test class. This template allows the test class author direct control over the C++ header generated for this test class, and should be between the keywords **CPlusPlusBegin** and **CPlusPlusEnd**. It uses various replacement indicators (such as **$ImportDeclarations**, **$ClassName**, **$ParamArrayTypes**, **$ParamFunctionDeclarations**, **$ParamAttributes**, **$ParamFunctionImplementations**, etc.) which specify the locations for various declarations.

### 5.3.1 Pre-Header Parameters Section

The **Parameters** section of an OTPL pre-header is used by the test class author to specify parameters for the test class. These are basically members of the test class that the test class user can specify in instances, providing finer control of the operation of the test. For instance, a test class author could declare a parameter TraceLevel, which can be specified by a test class user in order to control the level of tracing when the test executes.

There are three types of entities that can be declared within the **Parameters** section of a test class:

1. Enumerations using the **Enum** keyword.
2. Parameters using keywords such as **Integer**, **PList**, etc. specifying the parameter type, where the type is a valid OTPL type.
3. Groups of parameters using the **ParamGroup** keyword.

The **Enum** keyword is used to define a test class enumerated type. The test class can then have parameters of this enumerated type. This can also be used by a GUI to construct a drop-down list of the allowed values for the parameter.

Individual parameters of the test class can be declared within the **Parameters** section. Parameter declarations have a syntax which declares the type of the parameter, its

name, and then various properties of the parameter. Among the most important ones are the following:

- **`Cardinality`**: This indicates the number of parameters of this type that will be supported. The details of the allowable values are given in [15].
- **`Attribute`**: The name of the C++ variable to use as the store for parameter value(s) of this type.
- **`SetFunction`**: The name of the function to use for setting a value for this parameter. An optional keyword "**`[implement]`**" following the function name would indicate that a trivial implementation for this function will be made available as an inline method in the class header (inserted at the point indicated by **`$ParamFunctionImplementations`**). Otherwise, the user is responsible for providing an implementation of the function.
- **`Description`**: A string that can be used by a GUI to provide a tool-tip in a window that displays the parameter and its value. This description, along with other information about each parameter (such as its type, cardinality, and value), are made available to a GUI via a `getXMLDescription()` method (cf. Listing 3), which is described in greater detail below.

Finally, the **`ParamGroup`** keyword is useful to collect a number of different parameters as a group, similar to a C++ class.

### 5.4 XML Export of Test Class Parameters

Since test classes support class-specific parameters, no application will be able to know *a priori* all the parameters of all test classes. Therefore, to support all test classes in a generic fashion, the OTPL compiler generates the method "`OFC::String getXMLDescription()`" (see the example given in Listing 3). This function returns an XML description string. The description of the parameters for an instance of a class comprise information about each of the parameters of the class, and all classes of which it is directly or indirectly a subclass. This information includes name, data type, current value, application hints (e.g. multi-select), etc. The TSS tools *application programming interface* (API) allows for automatic parsing of this string using an XML parser, and for providing the resulting data to requesting applications.

The principal benefit of this approach to the TSS is that it provides self-describing data that is simple to generate using an open standard (XML). The data may be parsed by a number of standard tools available in several programming languages or using an easily written parser. XML allows existing applications to be free from side effects due to the addition of new attributes or elements in the description strings.

This concludes our discussion of the test programming environment in the TSS. In the next section, we describe the approach taken in the TSS for pattern management in an OA environment.

## 6 Pattern Data Management

In an open architecture system, in general, a single DUT might be connected to tester *modules* from different vendors. A *module* in this discussion refers to a subsystem in an ATE product, and can include both hardware and software components. This module participates in the framework provided by the TSS.

The user of the system writes a *pattern* focusing on the DUT, possibly without regard for the fact that this pattern may be compiled for multiple tester modules developed by different hardware vendors. This assumes, of course, that each of these modules is capable of providing the features specified in the pattern. This also includes the possibility that vendor-specific pattern formats can be included within the pattern file. This has implications for the entire pattern compile–load–execute chain. The various issues and how the TSS deals with them are described in this section.

### 6.1 Open Architecture Issues for Patterns

In the current generation ATE, patterns are described using a pattern language. There have been efforts to standardize this language so that patterns can be shared among different ATE systems. One example is the STIL effort [14]. However, there has not been much effort made in the direction of also standardizing the compiled object file. One of the main reasons for this has been that the generated object files are very closely tied to the vendor specific hardware and no standard has existed for this hardware. Another reason is that various vendors do not want to publicly share their proprietary formats. Enforcing a common format would also lead to compromises in the efficiency of pattern use.

### 6.2 The Concept of the Pattern Meta-File

In ATE systems, as the contents of a pattern object file are very closely tied to propriety hardware it is very difficult to come up with a general format. However, instead of regulating the lowest layer of the format, the TSS has promoted a *meta-file* concept. Traditionally, meta-files have been used in graphics to store and transport various formats of data that are interpreted by specific end-user applications. In a similar fashion, the TSS framework uses meta-files to store pattern data.

The TSS pattern object meta-file has provisions for storing

- common header information,
- module-specific header information, and
- module-specific pattern data, organized as required by the module vendor, and capable of being interpreted by the module vendor.

This approach satisfies the need for allowing various vendors to continue to use their propriety formats as well as being able to co-exist in an open framework. The vendor specific sections of the meta-file are treated as opaque binary data, about which the framework itself has no internal knowledge.

## 6.3 The Object File Manager

A vendor's compiler will not directly create the pattern object meta-file described in §6.2. Instead, the TSS provides an *Object File Manager* (OFM) framework that is instrumental in the creation of pattern object meta-files. The OFM framework manages the storing of common pattern data into the meta-file, but relies on plug-ins from various module vendors to invoke the compilation of vendor specific data. The model for the TSS pattern compilation process can thus be conceptualized as shown in Figure 2 below.

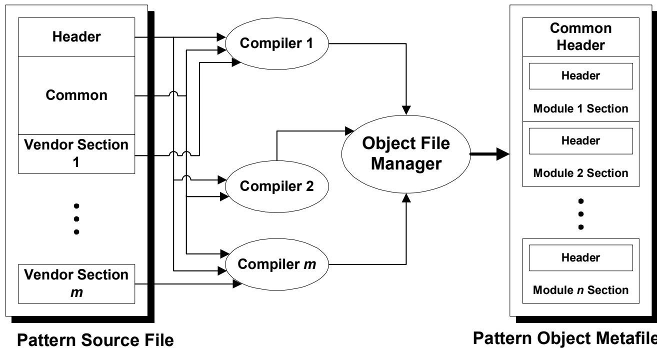


**Figure 2. TSS Pattern Compilation**

Each module vendor is responsible for providing their own pattern compiler, in the form of a dynamically loadable-shared library. This library is required to provide, at the very least, a complete implementation of the required OPENSTAR™ interfaces, so that it can function as part of the OFM framework.

The OFM framework also provides a set of interfaces designed to provide vendor compilers access to *common* data that are required to be able complete the task of pattern compilation. This includes access to run-time objects that encapsulate the description of DUT pins, DUT sockets and DUT timings, for example. It also provides the capability for accessing other referenced pattern meta-files, such as pattern subroutines.

The higher-level TSS pattern compiler tool is thus actually a composite pattern compiler. It orchestrates the compilation of patterns by itself reading various system configuration files, but depends on the various module compiler plug-ins to actually compile the vendor specific pattern data.

## 6.4 OFM and Vendor Compiler Interactions

The user invokes the compilation of a TSS pattern by supplying the filename of the pattern source file. The compiler instantiates a software entity (say, `PatternObjMetaFile`), representing the pattern object meta-file, with this name. If the user has specified an *overwrite* mode, the original contents of the meta-file, if any, are destroyed. Otherwise, when the OFM calls the `open()` method of `PatternObjMetaFile`, the *common* contents are uploaded from the meta-file on disk. The OFM then queries the system configuration files and adds/removes the appropriate vendor module sections in the meta-file.

After this, the OFM instantiates an object derived from the standard OPENSTAR™ interface `IVendorPatCompiler` for each vendor module, after loading that module's compiler library. An implementation of this interface is thus required to be provided by the vendor in a DLL.

The OFM then updates data in the *common* section by explicitly calling the respective methods in the `IVendorPatCompiler` interface. It makes use of these methods only for the first module's compiler. Vendor compilers need to provide support for these methods, and should be able to parse the source file and provide the queried data to the OFM. Subsequent module compilers can query the OFM to get access to the *common* data shared by the various modules.

The actual compilation of a module's pattern is accomplished by a call to the `compile()` method on the vendor's implementation of the `IVendorPatCompiler` interface. Vendor module compilers then use another standard interface (passed in with the `compile()` call) to access runtime information (such as pin information and timing data), as well as write their own specific pattern data to disk.

Note that the TSS pattern compiler tool coordinates access to common data such as pin definitions, socket definitions, timing data, etc., in such a way as to maximize efficiency, not requiring each vendor to re-generate the same data.

## 6.5 Pattern Loading for a Module

In the TSS, each module vendor is responsible for providing its own pattern loading mechanism. As seen in §6.2, the pattern object meta-file stores module-specific data in different sections. The vendor pattern loader implementation uses the TSS OFM APIs for accessing the relevant

sections from the pattern object metafile. The TSS framework is responsible for calling each module's *load* method in turn.

## 7 Summary

This paper described the support provided by the TSS for

- a modular, iterative development of user test programs,
- generalized user test object creation, where the system allows exactly the same methods to be used to create a plethora of parameterized, custom test classes that can be treated with exactly the same facility by the system, and
- pattern data management in a multi-module, multi-vendor open environment.

These are important factors in establishing the OPENSTAR™ system as a versatile open architecture platform for ATE.

Due to the very nature of the problems it addresses, the OPENSTAR™ system is vast, and a complete description of it is beyond the scope of this paper, which concentrated instead on a few particularly useful factors that contribute towards its goals of providing a flexible, modular, scalable and open test system. Work is currently in progress in making available a detailed specification of the entire OPENSTAR™ system. The interested reader is referred to [13] for full details.


## Acknowledgements

The authors are grateful to the engineers and management at Advantest America R&D Center in the USA, and at Advantest Corporation Gunma R&D Center in Japan, for making the T2000 system a reality (Leon Chen did this work while he was with the Advantest America R&D Center in Santa Clara).

OPENSTAR™ itself is being supported by an industry group of semiconductor and ATE manufacturers, united under the auspices of the Semiconductor Test Consortium, the STC. The authors wish to thank the STC members for the numerous discussions that have led to the development of the OPENSTAR™ specifications.


## References


[1] D. Mirizzi *et al*., "Implementation of Parallelsite Test on an 8-bit Configurable Microcontroller", *Proc. of the IEEE International Test Conference*, October 1993, pp. 226 – 235.

[2] U. Schoettmer and T. Minami, "Challenging the High Performance — High Cost Paradigm in Test", *Proc. of the IEEE International Test Conference*, October 1995, pp. 870 – 879.

[3] W. R. Simpson, "Cutting the Cost of Test: The Value-Added Way", *Proc. of the IEEE International Test Conference*, October 1995, pg. 921.

[4] E. Chang *et al*., "A Scalable Architecture for VLSI Test", *Proc. of the IEEE International Test Conference*, October 1998, pp. 500 – 513.

[5] A. Evans, "Application of Semiconductor Test Economics and Multisite Testing to Lower Cost of Test", *Proc. of the IEEE International Test Conference*, September 1999, pp. 113 – 123.

[6] J. Johnson, "ATE Open Architecture Initiative", *Intel Corporation white paper*, February 2002.

[7] D. Conti, "Mission Impossible? Open Architecture ATE", *Proc. of the IEEE International Test Conference*, October 2002, pg. 1207.

[8] B. G. West, "Open ATE Architecture: Key Challenges", *Proc. of the IEEE International Test Conference*, October 2002, pg. 1212.

[9] S. Perez, "The Consequences of an Open ATE Architecture", *Proc. of the IEEE International Test Conference*, October 2002, pg. 1210.

[10] R. Garcia and B. G. West, "Hardware Essentials for an Open Architecture", *EE-Evaluation Engineering*, October 2003.

[11] M. Gavardoni, "Data Flow Within an Open Architecture Tester", *Proc. of the IEEE International Test Conference*, October 2003, pp. 185-190.

[12] A. T. Sivaram *et al*., "XML and Java for Open ATE Programming Environment", *Proc. of the IEEE International Test Conference*, October 2003, pp. 793 – 800.

[13] The Semiconductor Test Consortium (STC), "OPENSTAR™ Specifications", *through http://www.semitest.org*.

[14] IEEE Standards Association, "Standard Test Interface Language for Digital Test Vectors", *IEEE Standard 1450.0-1999*, 1999.

[15] R. Krishnaswamy *et al*., "The OPENSTAR™ Test Programming Language", *Research Report*, Advantest R&D Center, November 2002.

[16] R. Rajsuman *et al*., "Open Architecture Test System: System Architecture and Design*", to be presented at the IEEE International Test Conference*, October 2004.

[17] Erich Gamma *et al*., "Design Patterns", *Addison-Wesley Publishing Company*, 1995.