# DATA TYPE EXTENSIBILITY IN AUTOMATIC TEST SYSTEMS

Ion A. Neag, TYX Corporation, (703) 264-1080, ion@tyx.com
Stefan Gal, TYX Corporation, (703) 264-1080, stefan@tyx.com
Danver Hartop, DSI International, (714) 637-9325, dhartop@dsiintl.com

## ABSTRACT

This paper highlights the importance of data type extensibility for preventing software obsolescence in Automatic Test Systems. It describes solutions proposed by the authors for supporting such extensibility in several interfaces of Automatic Test Systems. Some of these solutions are proposed for interface standards currently under development, while others are implemented in existing commercial products.

Keywords: Automatic Test Systems, ATS, architecture, obsolescence, data types, customization, extensibility

## 1 INTRODUCTION

The introduction of new technologies often requires hardware and software upgrades in the Automatic Test Systems (ATSs) used to test the new equipment. While hardware upgrades may be generally addressed by adding new instruments to existing modular test stations, software upgrades may involve significant changes in the existing ATS software, which may cause compatibility problems for Test Program Sets (TPSs) already in use.

This paper addresses an extensibility issue related to ATS software upgrades. The higher complexity of modern equipment and the expansion of diagnosis towards the system level require the processing, exchange and storage of *complex data.* Frequently, the new technologies use *new data formats or new types of signals*, such as: data packets transmitted via digital buses; video, infrared and laser signals generated and sensed by Electro-Optics (EO) equipment, etc.

Because the data formats or signal types required in future applications cannot be anticipated at the time the original ATS software is designed, such software should *support future extensibility for data types*. To minimize upgrade costs, the addition of new data types should be possible *without changing the system software* itself, via a "plug in" mechanism.

The authors propose several solutions for data type extensibility, applicable for different interfaces of the ATS software. The feasibility of these solutions is demonstrated via prototyping and/or implementation in Commercial Off-The-Shelf (COTS) products.

The following COTS products are referenced in the paper:
- *TYX TestBase* - a test executive that supports the visual display and run-time execution of test strategies. TestBase is able to control the execution of external test procedures

developed with various languages and environments, the display of documents and reports with different formats and the storage of test and diagnostic results in relational databases and XML files.

- *DSI eXpress* - a diagnostic development tool that supports graphical input of UUT diagrams, their functional characterization, testability analysis and generation of test strategies.
- *Racal Instruments newWave* - a test development environment based on the future Signal Definition and Test Description (SDTD) standard developed by the IEEE [8], supporting the visual definition and simulated generation of complex signals.

Section 2 of the paper identifies the requirements for data type extensibility, including the ATS interfaces where such extensibility is necessary. The next Sections contain the analysis of extensibility issues for different interfaces, as well as the description of particular solutions developed by the authors.

## 2 REQUIREMENTS FOR DATA TYPE EXTENSIBILITY IN ATS SOFTWARE

### 2.1 Use Of Parametric Data In ATS Software

Modular ATS software supports the transfer of *parametric data* between modules, as exemplified in Figure 1. The **Test Executive** module imports from the **Diagnostic Software** test strategy information, including test sequence information and test input parameters. The above data may be transmitted directly, through software component interfaces, or indirectly, via data files. The **Test Executive** controls the execution of **Test Procedures**, performed by various **Test Environments**, sending test input parameters and receiving test output parameters. The parameters may be displayed and edited by the **User**. Parameters may be stored persistently in the **Test Data Store**, which can be a database or a data file.
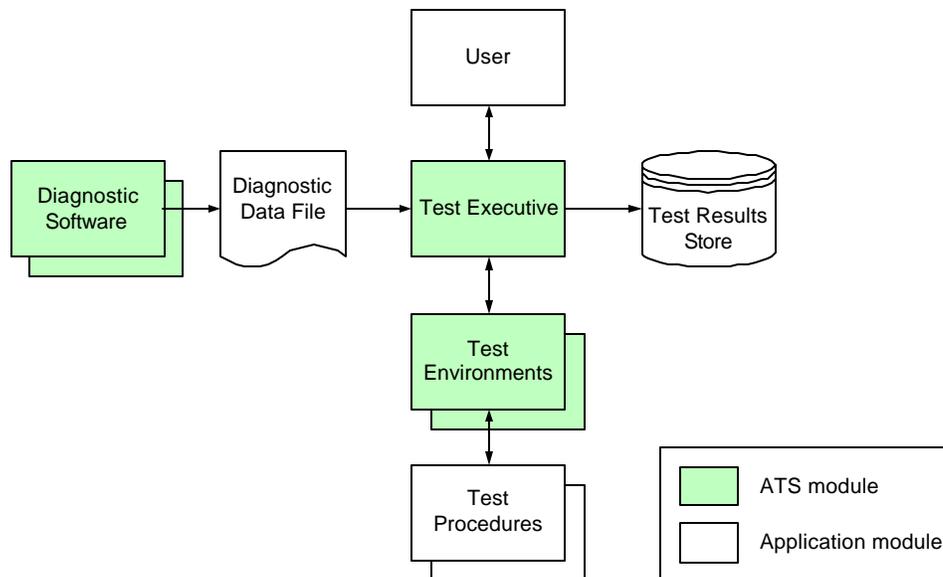


**Figure 1.** Parametric Data Transfer in an Automatic Test System - Example

In conclusion, data type extensibility is necessary in the following *categories of ATS interfaces*:
1. Software component interfaces.
2. Data formats, including databases and data files.
3. Graphical user interfaces.

A special case of extensibility in ATS software is encountered in signal-based test environments, such as those based on the ATLAS language or the SDTD standard. Signal-based test programs rely on a set of precisely defined signal types ("nouns" in ATLAS), each of them characterized by a set of signal parameters (ATLAS "modifiers"). As indicated in Section 1, new technologies involve the use of new signal types, requiring the signal-based ATS software to support the extensibility of signal types. While such extensibility is built in the ATLS language, it must be also supported in the SDTD signal libraries and the software interfaces of signal drivers.

## 2.2 Data Types

Although it is difficult to anticipate the data types required by future applications, the past and current use of data types in general-purpose software development indicates that *arbitrarily complex data structures* may be built using the following data types:
1. primitive data types: boolean, character, various integer and floating-point types
2. strings
3. records: ordered sets of fields that may have different data types (for example, the `struct` data type in C)
4. arrays, where all elements have the same data type

Arbitrarily complex data structures may be obtained if record fields and array elements are allowed to contain records or arrays.

## 3 DATA TYPE EXTENSIBILITY IN SOFTWARE COMPONENT INTERFACES

The extensibility mechanisms usable in software component interfaces depend on the interfacing technology. The component technologies most frequently used in ATS software are Dynamic Link Libraries (DLLs) and the Component Object Model (COM).

DLL interfaces may be implemented with Microsoft Visual C++, LabWindows/CVI and other ANSII C compilers and may be used from a large number of programming languages and test environments. DLL interfaces are standardized for VISA, VXI*plug&play* drivers and IVI Drivers. DLL interfaces support the transfer of complex data structures defined using the C data types `struct`, `union`, string and array. Alternatively, applications that run only on the Windows platform may use the Windows-specific data types `VARIANT`, `SAFEARRAY` and `BSTR` [2], which simplify dynamic resizing and memory management.

COM interfaces may be implemented with Microsoft Visual Basic and Microsoft Visual C++ and may be used from a large number of programming languages and test environments. COM interfaces are standardized for VISA and IVI Drivers. COM interfaces may use the Windows-specific data types introduced before, or object structures based on collections.

The next section shows an example of data type extensibility and signal type extensibility in a COM interface.

## 3.1 IVI Signal Interface

The IVI Foundation is currently working on a specification for the IVI Signal Interface (IVI-SI), an instrument control interface based on the signal abstraction [6]. The IVI-SI is applicable to the signal drivers used in ATS software based on the ATLAS language and on the new SDTD standard. This section describes a solution for signal type and data type extensibility, proposed by the authors for the design of the IVI Signal Interface.

To enable signal type extensibility, the definition of the interface must be independent of any particular signal type definitions. Consequently, the methods of the interface must have generic arguments, which support the transmission of signal parameters for any signal type. In addition, the interface design must support arbitrarily complex data types for signal parameters.

In the proposed solution, the parameter values of an arbitrary signal are stored in a data structure defined as follows:
- The values for a set of signal parameters are stored in an **IviSigParamSet** object, which is a COM collection of **IviSigParam** objects.
- Each **IviSigParam** objects has a Name property of type BSTR storing the parameter name and a Value property of type VARIANT storing the parameter value.
- Each Value VARIANT contains one of the following types of data:
  - For parameters with scalar data types (such as boolean, long, double, etc.), the value is stored in the corresponding field of the VARIANT (boolVal, lVal, dblVal, etc.).
  - For parameters with string data types, the value is stored in a BSTR data structure, whose pointer is stored in the bstrVal field of the VARIANT.
  - For parameters with array data types, the value is stored in a SAFEARRAY data structure, whose pointer is stored in the parray field of the VARIANT.
  - For parameters with record data types, the value is stored in an **IviSigParamSet** collection, whose COM interface pointer is stored in the punkVal field of the VARIANT. The **IviSigParam** objects in the collection represent record fields, with the field name stored in the Name property and the field value stored in the Value property.

Because array elements and record field values are of type VARIANT, they may hold any of the above data types, allowing the definition of arbitrarily complex data structures.

Example 1: The setup of a distortion measurement for an infrared sensor requires the following data items (only a limited sub-set is described, for illustration purposes):

- measurement method (with the possible values "narrow" and "wide")
- differential temperature (`double`)
- an arbitrary number of test points, where each test point is characterized by:
  - azimuth angle (`double`)
  - elevation angle (`double`)

The parameters of an "Infrared" signal are represented by the data structure shown in Figure 2, where the SAFEARRAY data structure has one dimension and contains one element for each test point.
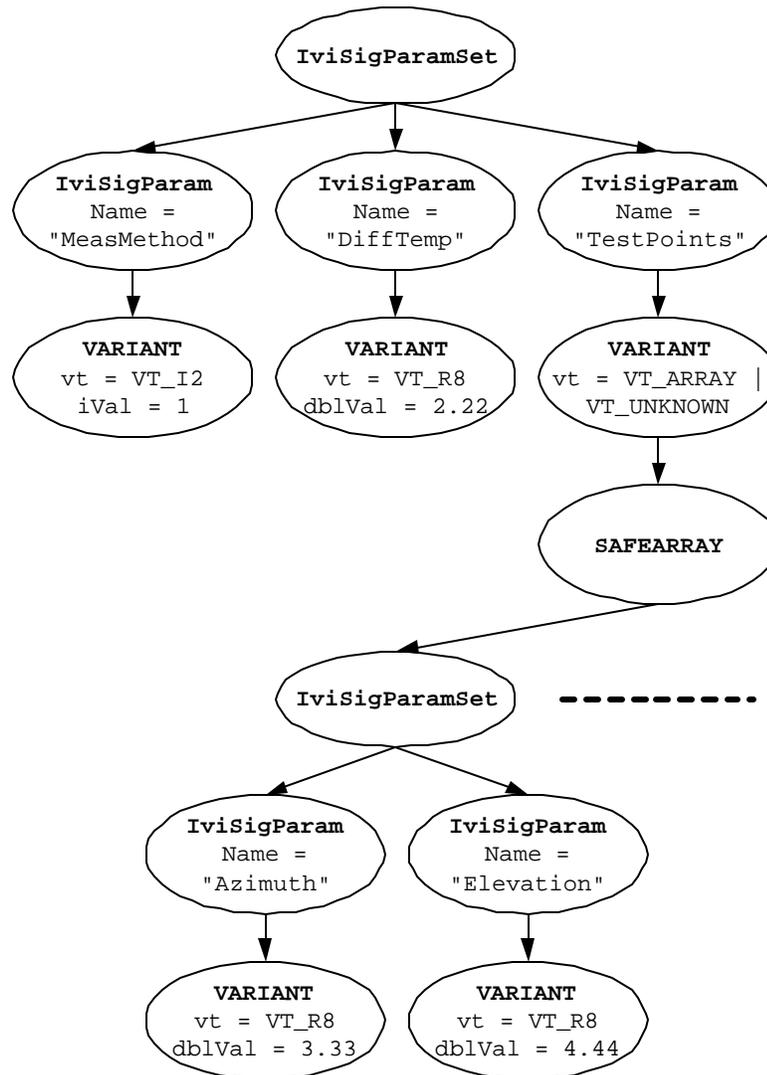


**Figure 2.** COM Object Structure for an "Infrared" Signal Type - Example

# 4    DATA TYPE EXTENSIBILITY IN DATA FILE FORMATS

Data files may be used to exchange information between software modules developed with a variety of software technologies. File formats are defined by standards such as AI-ESTATE [1], TEDL, EDIF, DTIF, etc.

Traditionally, file formats have provided limited extensibility for data types. The use of the Extensible Markup Language (XML) [3] for data exchange has the potential to overcome this limitation. XML-based formats are defined using XML schemas, which support a variety of predefined data types and enable their composition in arbitrarily complex data structures [4]. Alternatively, such data structures may be serialized as hex-encoded binary data, using the primitive data type `hexBinary`, defined by the XML Schema specification [4].

The next section shows an example of data type extensibility in an XML-based data file format.

## 4.1    DiagML

DiagML [7], the Diagnostic Modeling Language, is an XML-based, non-proprietary language developed by a consortium of companies, supporting the exchange of diagnostic and test data between diagnostic software and test executives developed by different vendors.

DiagML supports the definition of complex data types by combining simple types in "arrays" and "groups" (DiagML-specific name for records). The XML schema definitions for *DiagML value*, *DiagML primitive value*, *array* and *group* are shown in Figure 3. A *DiagML value* may be a *DiagML primitive value* or a *DiagML custom value*. The role of DiagML custom  values is described in Section 6.2. A *DiagML primitive value* may have one of the primitive data types supported by the XML schema, such as *long*, *double*, *string*, etc. (only a subset of the supported data types being represented), or may be an *array* or a *group*. An *array* element holds the *dimensions* of the array (i.e., numbers of elements over each dimension) and the *values* of array *elements*. The *group* is an ordered set of *value* elements, the corresponding record field being identified by the position of each element.

*Values* stored in *arrays* and *groups* may have any of the above data types, including *array* and *group*, allowing the definition of arbitrarily complex data types.
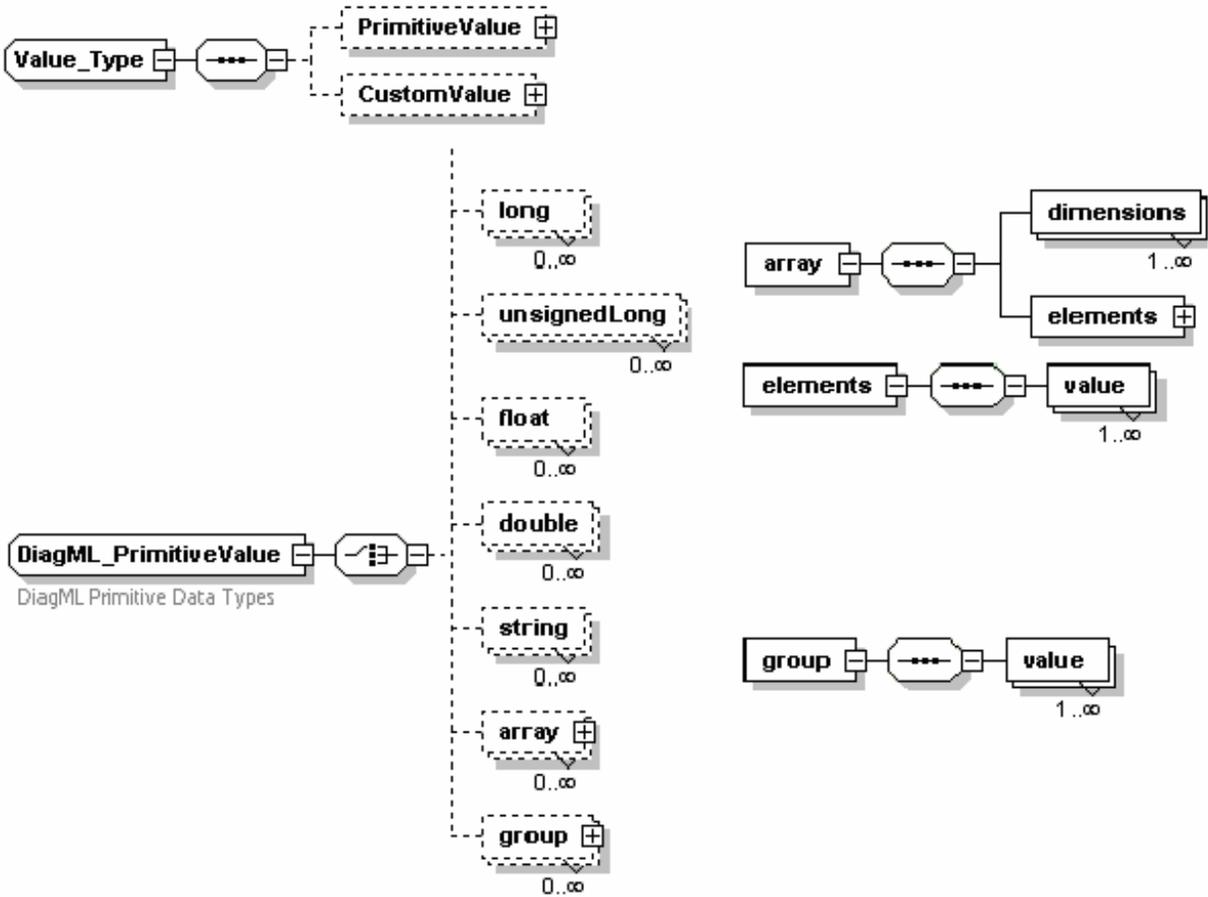
**Figure 3.** Data Type Extensibility in DiagML

Example 2: The input parameters of a "distortion measurement" test, introduced in Example 1, may be represented in DiagML as shown below.

```
<ParameterValue> <!-- for "MeasMethod" -->
  ...
  <Value>
    <PrimitiveValue> <short val="1" /> </PrimitiveValue>
  </Value>
</ParameterValue>

<ParameterValue> <!-- for "DiffTemp" -->
  ...
  <Value>
    <PrimitiveValue> <double val="2.22" /> </PrimitiveValue>
  </Value>
</ParameterValue>
```

```
<ParameterValue> <!-- for "TestPoints" -->
  ...
  <Value>
    <PrimitiveValue>
      <array>
        <dimensions>
          <nelements> 3 </nelements>
        </dimensions>
        <elements>
          <group> <!-- for 1st test point -->
            <Value> <!-- for "Azimuth" -->
              <PrimitiveValue> <double val="3.33" /> </PrimitiveValue>
            </Value>
            <Value> <!-- for "Elevation" -->
              <PrimitiveValue> <double val="4.44" /> </PrimitiveValue>
            </Value>
          </group>
          <group> <!-- for 2nd test point -->
            ...
          </group>
          <group> <!-- for 3rd test point -->
            ...
          </group>
        </elements>
      </array>
    </PrimitiveValue>
  </Value>
</ParameterValue>
```

## 5   DATA TYPE EXTENSIBILITY IN DATABASE FORMATS

Test and diagnostic results are frequently stored in relational databases. In such databases, arbitrarily complex data types may be serialized in variable-size binary fields, commonly called BLOBs.

The above approach is used in the Maintenance Test Information (MTI) database of TestBase [MTI ref]. A software component distributed with TestBase implements the de-serialization of binary values retrieved from the database, supporting the development of custom applications that use MTI data.

## 6   DATA TYPE EXTENSIBILITY IN GRAPHICAL USER INTERFACES

Parametric data are entered by developers via the Graphical User Interfaces (GUIs) of test development environments and are typically displayed to operators through GUIs specialized for the production environment. In order to support data type extensibility, the above GUIs must provide a *generic parameter editing/display* capability, or enable the integration of *custom data editors/viewers*.

The next section describes the design of custom data type editors (CDTEs) supported by DSI eXpress and TYX TestBase.

## 6.1   Custom Data Type Editors

The CDTEs supported by eXpress and TestBase are <mark>ActiveX controls</mark> providing the following interfaces:

- A graphical interface that supports the display and editing of data
- A generic COM interface that enables the transfer of data to and from the host applications
- A specific COM interface that enables access to data stored by the CDTE

The graphical interface is specific for each custom data type, while the COM interface used for interaction with the host application is generic. This feature enables the "plug-in" integration of CDTEs, without modifying or even re-compiling the host application. With this approach, a new CDTE installed on the system becomes instantly available to both eXpress and TestBase. Thus, CDTEs may be developed by end-users or system integrators, extending the native support provided by eXpress and TestBase for specific application domains.

For instance, a set of re-usable CDTEs may be developed for application domains such as Radio-Frequency (RF), Electro-Optics (EO) or digital bus testing. An RF-specific CDTE may support the graphical definition of power gain limits. Moreover, CDTEs may support the import of parametric data from third-party applications. For example, the above CDTE may import data from Microsoft Excel worksheets where power gain curves are calculated through mathematical formulas.

<u>Example</u>: An EO-specific CDTE supporting the input parameters of a "distortion measurement" test, introduced in Example 1, may have the graphical interface shown in Figure 4.

<mark>TBD – from NxTest pres. slides</mark>

**Figure 4.** Graphical Interface of a Custom Data Type Editor

The use of a CDTE in the integrated eXpress -TestBase architecture is illustrated in Figure 5. When the user edits a test parameter whose data type is supported by a CDTE, the graphical interface of the CDTE is automatically displayed by eXpress or the TestBase Integrated Development Environment. When the user finishes the editing operation, a serialized form of the edited value is returned to the application via the generic COM interface of the CDTE. eXpress stores the serialized values internally and exports them to TestBase as parametric data associated with test strategies. TestBase stores the serialized values internally and transfers them at run-time to the test procedure code. The test procedure code instantiates the CDTE component without displaying its graphical interface, passes the serialized value via the generic COM interface and retrieves the de-serialized data via properties exposed by the specialized COM interface.
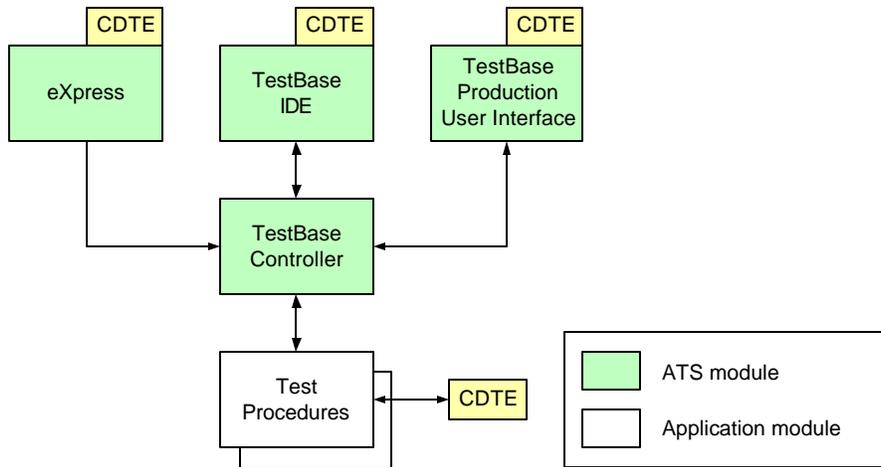
**Figure 5.** Use of Custom Data Type Editors in eXpress and TestBase

Parameters with custom data types are displayed by the TestBase Production User Interface using the same CDTE components, operating this time in read-only mode.

## 6.2 Supporting Custom Data Type Editors in DiagML

eXpress exports of test strategy data to TestBase using the DiagML format (see Section 4.1). The transfer of custom data types between eXpress and TestBase is supported by a special feature of DiagML. As described in Section 4.1, a *DiagML value* may be a *DiagML primitive value* or a *DiagML custom value*. The *DiagML custom value*, whose schema definition is shown in Figure 6, contains a *serial data* element storing the hex-encoded binary data and a string element storing the COM Programmatic Identifier (*ProgID*) of the CDTE used to edit the value in eXpress. The storage of the ProgID in the DiagML file guarantees the use of the same CDTE to access the value, after it is imported in TestBase.
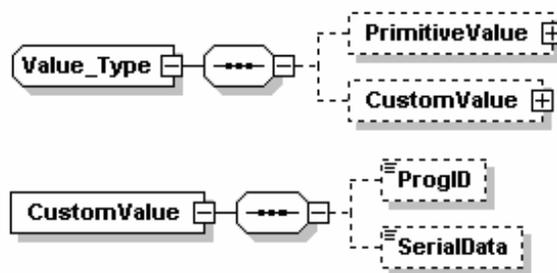


**Figure 6.** Support for Custom Data Type Editors in DiagML

## 6.3 Application Integration via Custom Data Type Editors

This section describes another possible use of CDTEs - integration of third-party applications in TestBase and eXpress.

The *newWave* application developed by Racal Instruments, which supports the graphical definition of complex signals using SDTD diagrams, may be integrated in TestBase through a CDTE. This solution enables the graphical definition of signals to be passed as input parameters to test procedures. The architecture supporting the integration solution is illustrated in Figure 7.
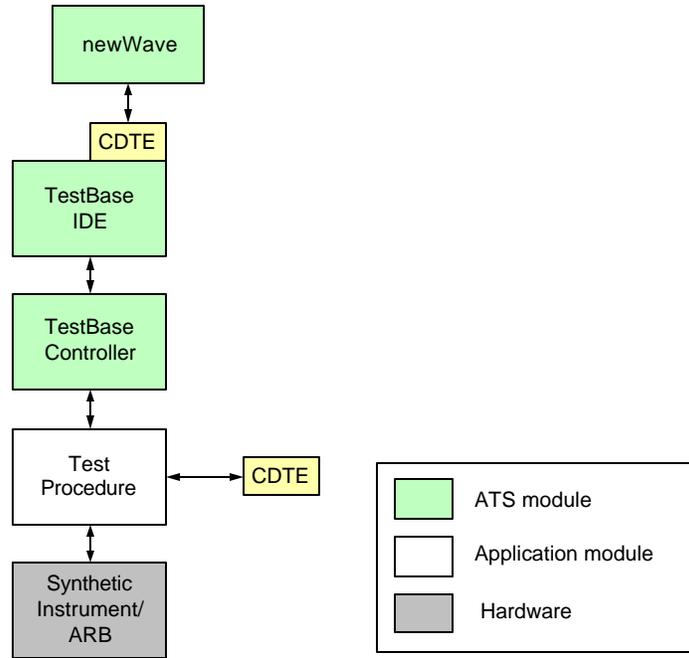


**Figure 7.** Graphical Signal Definition via newWave - TestBase Integration

A specialized CDTE module, integrated in the TestBase IDE, supports the interaction with *newWave*. The graphical user interface of *newWave* allows the developer to define signals using SDTD diagrams. In addition, *newWave* calculates and displays signal samples. When the editing of a signal is finished, the CDTE returns to TestBase, in a serialized format, the SDTD definition of the signal (necessary) and a set of signal samples with the associated timing. The SDTD definition is stored by TestBase and may be used for editing the signal definition at a later time. The signal samples, also stored by TestBase, are transmitted at run-time to the test procedure that controls the generation of the signal. The test procedure code uses the same CDTE component to convert the serialized data in signal samples, and then sends the samples to an arbitrary waveform generator (ARB) or a synthetic instrument that performs the generation of the physical signal.

The feasibility of the previously described integration is demonstrated by a prototype developed jointly by TYX and Racal Instruments.

## 7   CONCLUSION

The requirements for data type extensibility must be taken into account in the design of ATS software in order to prevent software obsolescence. The implementation of such extensibility is supported by the software technologies used in the development of modern ATS software.

The solutions described in this paper demonstrate data type extensibility and signal type extensibility, implemented in all the relevant ATS software interfaces.

## 8   REFERENCES

[1]         *Draft Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)*, IEEE Standards Coordinating Committee 20 on Test and Diagnosis for Electronic Systems, IEEE, 2002

[2]         *Windows API*, Microsoft Developer Network Library

[3]         *** *Extensible Markup Language (XML)*, World Wide Web Consortium (W3C), http://www.w3.org/XML/

[4]         *** *XML Schema*, World Wide Web Consortium (W3C), http://www.w3.org/Schema/

[5]         Neag, Ion A., Ramachandran, N, "ATLAS2K and the IVI Signal Interface - the Framework for an Open, Modular and Distributed ATS Architecture", *2001 AUTOTESTCON Proceedings*, pp. 23 - 37, IEEE, 2001

[6]         *** *IVI Signal Interface Working Group*, IVI Foundation, http://www.ivifoundation.org/groups/Signal-Interfaces/default.htm

[7]         *** "What Is DiagML", *DiagML Web Site*,


[8]         Ellis,K., Delaney, D., Gorringe, C., "ATLAS2K – The Framework for Precise and Extensible Signal Descriptions in Modern ATS", *2001 AUTOTESTCON Proceedings*, pp. 468 - 479, IEEE, 2001