

# XML And Java For Open ATE Programming Environment

A.T.Sivaram, Daniel Fan, Jon Pryce  
NPTest  
150 Baytech Drive  
San Jose, CA 95134

## Abstract

*Data flow based software architecture has been in use in the ATE environment for the last decade. Major ATE vendors typically provide the test engineers a database approach for representing test data and a programming language with tester specific extensions for implementing test methods. The data base implementation and tester extensions were based on custom syntax making the whole test program development environment coupled tightly with the closed ATE hardware architecture. This paper describes the implementation of test data using XML, a license-free, platform-independent and well-supported language, and implementation of test methods using Java, an open and platform independent interpretive language in a next generation architecture ATE platform. The advantages of this environment in extensibility, productivity and reliability for the ATE user are presented.*

## 1 Introduction

With the advent of the color graphics engineering workstation in test equipment during the mid 1980's ([1]) the procedural based execution of test programs gave way to flow based execution. The graphics workstation displayed test conditions and results, in a GUI environment [2]. In this graphical environment test and product engineers edited test conditions interactively to create, modify and debug tests on the Device Under Test and were able to cut down on the total development time for the test program. The test data was stored in structured blocks, and the test algorithm was written in a standard programming language such as C or C++ [3]. In today's ATE environment built-in test methods are included in the test system software for standard functional and parametric tests further reducing test program development time. What we have available today are testers from several different manufacturers with different GUI environments, proprietary block structure/database definitions and different application environments for implementing device tests.

With the explosion of System on a Chip devices fueled by silicon integration on smaller line widths, today's ATE are challenged to become multi functional testers. In the not-too-distant future the ATE landscape will be made up of testers that bring

together a multitude of instruments with different test capabilities integrated in a test head testing different blocks of an SoC device. Several ATE manufacturers are moving toward platform architectures that enable third parties to both co operatively and independently develop and market instrumentation options. In order to supply instruments to this market the software architecture must allow extension for new instruments to be implemented without a major effort expended for coding and making a new release. This paper describes a new software implementation for representing test program data blocks using EXtensible Markup Language (XML), a World Wide Web Consortium standard. XML provides a simple syntax using tags and data for representing block structures. Being closely related to HTML it has a textual format ideally suited for network friendliness. This new representation for test program structure forms the basis of the new software architecture, running on a PC platform, implemented in the next generation tester from NPTest. Section 2 presents an overview of the block structure used in the old tester architecture and discusses the challenges faced by this implementation in today's web based software environment. This is followed by the new software architecture description for the test blocks in section 3, showing the areas of improvement over the older architecture. Section 4 goes on to describe how the new implementation, using XML, provides for easy extensibility for block structure enhancement and new block additions to the software.

## 2 Background

The current generation software ASAP<sup>®</sup> (Advanced Symbolic ATE Programming) has been in use in the ITS9000 family of testers since 1990. The approach used is a hierarchical data structured method whereby the pertinent data describing the tests is defined in highly structured blocks of data, rather than by independent procedure calls. The philosophy behind this approach is to separate the data, from the program flow, into easy to manipulate structures. As such, they may be easily changed or moved around. Careful thought went into the content of these blocks to permit the majority of test programming activities to be done completely with these data constructs together with built in test methods in a run time library, with only very special activities causing the need to drop down into the

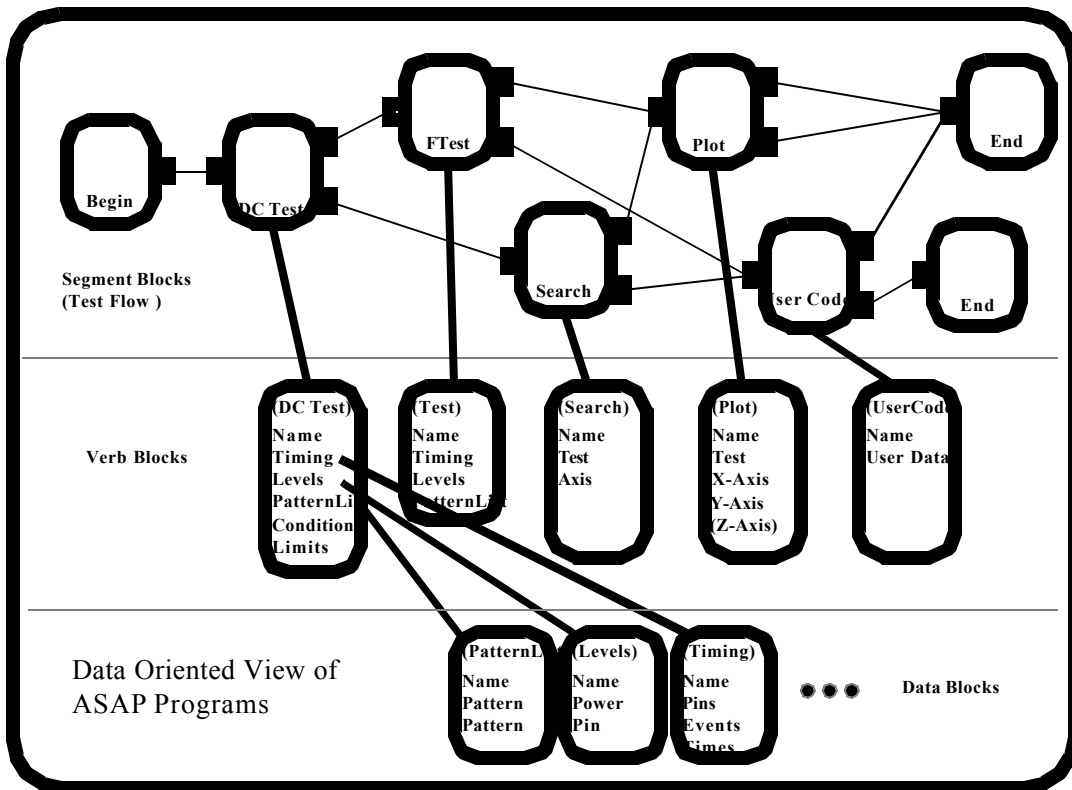


Figure 1. ASAP Block structure illustration

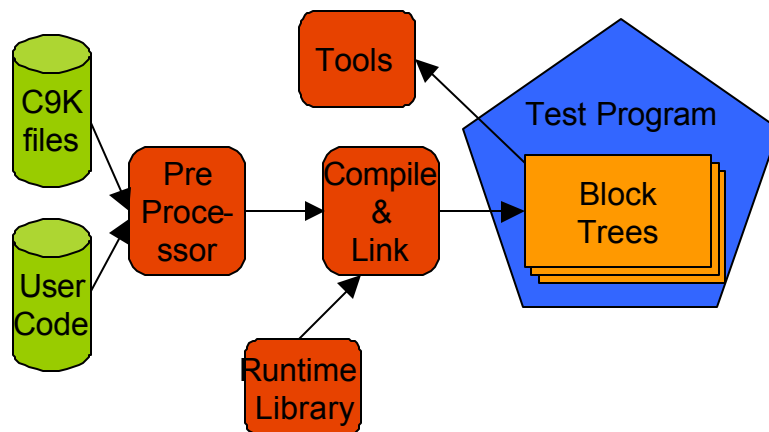


Figure 2. Test program environment under ASAP

writing of code. Blocks are used in a hierarchical fashion to decompose what has classically been a procedure-oriented solution into a clearly ordered data-oriented solution. This includes sections of code written by test programmers to accomplish custom test methods using 'C' functions which are compiled code. Also a block of data corresponding to each 'C' function is created which contains descriptions of arguments and a pointer to the code. With this structure in place, debug tools can synthesize calls to these functions and freely intermix them with the execution of tests when changing or creating test flow. There are two basic kinds of blocks. Action blocks cause something to happen and information blocks contain data to be used by the action blocks. The things comprising action blocks are tests, searches, plots and user tests. The data found in information blocks specify pin definitions, timing, levels, vectors, output care conditions and search or plot boundaries.

Another characteristic of the implementation of all blocks is that they contain their name as an ASCII string and pointer fields so that they can be put into a binary tree structure. A tree is constructed at install time for each kind of block. This facilitates the ability of debug tools to quickly access, display, modify and save any aspect of a test program without any compilation step.

The uppermost element of the hierarchy is the SEGMENT block. This is the block type with which a directed graph is created to specify what action blocks are to be called in what order. A SEGMENT block tells which action block should be called and which SEGMENT should be used next in the event of a pass or fail result. Action blocks refer to information blocks which contain the parameters required to do a DC TEST, FUNCTIONAL TEST, SEARCH, PLOT or a user written FUNCTION. Figure 1. illustrates the block structure nature, discussed thus far, of the test program pictorially.

Figure 2 shows the test program environment under ASAP. The flow and the tests are defined using segment blocks inside a source file, with extension c9k, which is first preprocessed by a custom preprocessor, cc9k, into 'C' type structures. The preprocessor also processes any custom user tests to create function block structures. The resulting 'C' code and data are compiled and linked, with a run time library consisting of built in test methods, using a 'C' compiler. When the resulting binary program is loaded and run the test engineer interacts with a set of GUI tools through the run time library to edit, modify, create, delete, duplicate and execute the blocks. The ASAP software also provides for saving the modified blocks back to the source format.

## 2.1 Attributes common to all blocks

- The skeleton structure of a block is

```
<block_type> block_name {
    <block data>      };
```

- In general, sections of a block that are part of a

keyword statement of the form  
KEYWORD = data

may be listed independent of order. That is if a particular block is composed of a series of keyword statements, they may be listed in any order.

- Sections of a block that are pure data (such as the data portion of a keyword statement) have a dependency on the order in which they occur in the source. For example, the values listed in a TIMING block will be programmed into the hardware in the order in which they are listed in the source.
- Data sections of a keyword statement often allow one or more like parameters to be specified. When this is the case, a single parameter may occur immediately after the keyword. When more than one parameter is specified, the list of parameters must be enclosed in braces "{...}". A single element may appear with or without braces.
- Comments may be freely intermixed anywhere within the source code by delimiting then with the standard 'C' comment syntax "/\* comments \*/".
- Many blocks, such as PINDEF, LEVELS, TEST, COMPARE, TIMING, DCTEST, SEGMENT, PLOT, ACTEST, and SEARCH, may also specify a revision date of the form "<mm:dd:yyyy hh:mm:ss>" that is retained with the block for revision control. If not specified, the date defaults to zero.
- There are many <block\_type>s in each of which the block\_type keyword in lower case is a reserved identifier, and may not be used in any other way. Each block is given an unique block name which is used to refer to all it's contents both in the test program and in the environment of the debug tools.

## 2.2 Current Architecture Challenges

The ASAP test program environment using the block structured approach presented thus far has been in use for more than a decade across five generations of the ITS9000 family of testers. It has gone through a few major revisions as well as several minor ones during this period to accommodate new tester capabilities, newer options to existing testers, software enhancements and defect fixing. A typical enhancement to an existing tester such as a source synchronous hardware capability would mean changes at the timing and levels blocks structure at the very least, addition to the run time test methods to comprehend source synch. testing and modification to the existing timing and levels tool GUIs. All of these are proprietary architectures owned and maintained by a single vendor. This option would mandate a new release of software to the customer base planning to acquire the new option. This mode of operation was definitely acceptable in prior years. However the proliferation of the world wide web has brought about major changes on how software is created and distributed. The best example for this is the process one goes about adding a new printer to a PC. The printer driver, if not already

on the PC, can be downloaded from the manufacturer over the network after which it will install itself in a matter of minutes. Once the test page is successfully printed the printer is ready for use.

For the next generation tester architecture it is obvious that the software environment must provide a level of flexibility the PC users already have today. What this means is that the architecture must use established standards and be release independent. In the next section the new architecture based on XML and Java is described in detail.

### 3 The Programming Model

#### 3.1 XML

XML was chosen to represent the data blocks in the new software architecture mainly for the following reasons.

- Simple syntax for structured data

As shown earlier the block structure used in ASAP consists of attributes and value pairs. XML allows a similar definition by use of tags (identifiers enclosed in angle brackets, like this: <...>). and data corresponding to each tag. Collectively, the tags are known as "markup". As the example below shows the attributes of the existing block structure definition can be directly used as tags in the XML .

```

Levels std {          <DCLevels name="std">
  PIN_LEVELS {      <PinLevels
    All_In = {      <sigref="All_In">
      VIL = 0V,    <VIL>0V</VIL>
      VIH = 4V    <VIH>4V</VIH>
    }
  }                </PinLevels>
/*Other levels*/   <!--Other levels-->
}                  </DCLevels>

```

- Extensible and easy for computer processing

XML has been designed to put structured data in a text file. XML allows the designer to define a set of rules, guidelines and conventions for designing text formats for data, in a way that produces files that are easy to generate and read (by a computer), that are unambiguous, and that avoid common pitfalls, such as lack of extensibility, lack of support for internationalization/localization, and platform-dependency. [4]

- XML is a World Wide Web Consortium open standard since Feb 1998 (revision 1.0). [www.w3.org/XML](http://www.w3.org/XML)

XML is license-free, platform-independent and well-supported. By choosing XML as the basis for test data representation a large and growing community of tools is available for use in addition to software engineers experienced in the technology. By opting for XML the software architecture enables third party vendors to easily integrate their test equipment compatible with our own database and our own programs/procedures that manipulate it.

- Allows easy implementation of STIL definitions into the test program environment.

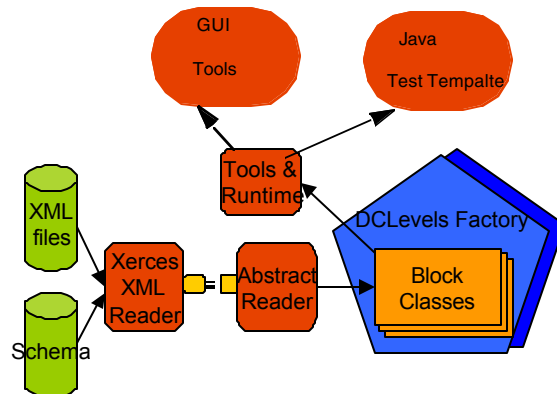
STIL the IEEE 1450-1999 standard [5] has been adopted as our native pattern language. STIL is still evolving in the area of test program content such as levels, test flow, binning and test methods. The DCLevels block already a new STIL standard has been adopted in the XML definition. Further for choosing the XML tags for other equivalent ASAP blocks the STIL naming and structure has been followed as closely as possible. Also STIL is quite close to ASAP from its inception.

#### 3.2 New Block Architecture

The overall software architecture consists of a base class called "GenericBlock" that handles all block types, although blocks that require special actions may inherit from it and override some of its methods.

During program start-up an external "ProgramLoader" class locates all files with a .xml extension in the test program directory, and calls the Abstract XML reader for each file found. For every top-level block found, the reader asks the block "factory" that knows about this type of block to create a new block object. The new object is then placed in a container reserved for this block type. The GenericBlock interface is then used to fill the data from each XML element found in the source file. The block data is

Figure 3. Test program environment under XML



stripped of the XML decoration by the reader and becomes effectively language independent. This allows the source language to be changed simply by the addition of a new type of reader. The GenericBlock class hierarchy manages the language block data and provides interface methods for access to the data. Each top-level XML block will have a corresponding GenericBlock. There is also a derived class of GenericBlock to store the Block Definition XML blocks that define the syntax and semantics of each language block type. GenericBlock instances are stored in a factory, and there is one factory per block type.

#### 3.3 Java Test Template

Figure 3 is a pictorial representation of the test program environment described thus far. The XML reader Xerces is a free package from Apache. The GUI tools communicate with the runtime to allow the user to perform the normal create, modify and update activities on the test program to enable device debugging. The test templates, which are user defined or system provided test methods, are implemented using Java. The Java Virtual Machine employs "Runtime Dynamic Binding". What this means is that when a method or field of a class is accessed, its address is determined by the JVM at the time of the first access. This contrasts with C and C++ which determine addresses and offsets within structures/classes at compile time. The disadvantage of this "Compile-time Binding" is that, if the data structure is later changed in a new software release for example by inserting a new field, all code which uses it must be recompiled. Both C++ and Java are object oriented languages, so are all the methods which are part of the class definition. Even adding a new method will force a C++ user to re-compile the program. Using Java provides for release independent test program generation. It also improves productivity since runtime errors such as "Null pointer referencing" and "out of bounds array access" which are commonly not reported during run time in a C++ compiler oriented environment are reported immediately in a Java environment. Java has many advantages, but it also has a known disadvantage. Java performs slower than C++, especially when it comes to I/O operations. Java may be slower, but, from the users perspective, there is relatively little difference. The advantages to Java outweigh its disadvantages in most cases. The ease of development, reusability and portability make Java attractive for the ATE environment as well. The performance difference can be reduced with the Just-in-Time compiler provided in Java. This Java Compiler produces code which is comparable with C++ compiled code execution as shown in Table 1. In this comparison an FFT al-

FFT complex (Direct + Inverse) CPU vs Nbr Points			
N	1024	32768	524288
Log2(N)	10	15	19
Java (with JIT) - ms	0.638	48.34	3543
Cpp - ms	0.41962	47	4320.3

Table 1. Execution time Java vs. C++

gorithm was executed for different array sizes ranging from 1K to half a million. The Java, with Just In Time compiler, execution times compare very favorably with C++ execution times for large array sizes.

### 3.4 XML Block Syntax

A number of block types exist in the new software architecture. these are very similar to the ones implemented in the current ASAP environment but have been designed to include STIL standards wherever applicable. Some example blocks are DCLevels Block, Timing Block, Signals Block, SignalGroups Block, NameMaps Block, Test Block and

Flow Block. XML schema has been used to define the block syntax instead of Document Type Definition, DTD, due to the capabilities offered by it and the wide support it has. The general attributes of the XML block definitions are:

- The lowest level XML element is a single tag, value pair. E.g.: <Period>1.0e-8</Period>
- The element's value string is normally a single logical value rather than a list - lists are split into separate XML elements so that they can be accessed independently.
- XML subsections occur within the top-level XML element, nested to any depth. E.g.:

```
<Timing name="nominal_timing">
...
    <Waveforms name="seqA">
...
        </Waveforms>
    <Waveforms name="seqB">
...
        </Waveforms>
</Timing>
```

- The top-level tag always has a name attribute associated, unless there can be only one. A sub-section tag also normally has a name or other unique attribute to allow it to be distinguished from other sections of the same type, as in the <Waveform> sub-sections in the example above. Sub-section tag names may be omitted when at most one instance can appear in the block (in which case there is not much point in making a sub-section of it), or if random access to the section is not required (i.e. usage is always iteration through all such sections). Tag names are all lower-case.
- Extended character sets are supported for the value strings, but the XML tags themselves will conform to 7-bit ASCII encoding. For example this allows accented characters in user identifiers
- The value of all elements capable of holding a numeric value is passed to the equations package in case they contain expressions.
- Blocks can be written in any number of files. The order of blocks within a file is not important. There is no restriction on the number or type mixture in any one file. The files must however have a .xml extension.

#### 3.4.1 Example DCLevels source block

The example DCLevels block is simple. This is the equivalent of the ASAP Levels block. The XML tags reflect the draft STIL standard as closely as possible. This block defines appropriate levels for two signal pin groups and one power pin. The schema for the DCLevels block is shown Appendix A.

```
<DCLevels name="I8000MT_normal">
    <PinLevels sigref="data_bus">
```

```

<VIH>3.5 + (3.5*margin)/100</VIH>
<VIL>0</VIL>
<VOH>2.0 + (3.5*margin)/100</VOH>
<VOL>750E-3</VOL>
<ClampHi>vmax+0.5</ClampHi>
<ClampLo>vmin-0.5</ClampLo>
</PinLevels>
<PinLevels sigref="serial_if">
  <VIH>+2.8</VIH>
  <VIL>-2.8</VIL>
  <VOH>0</VOH>
  <VOL>0</VOL>
</PinLevels>
<PowerLevels sigref="VCC">
  <VForce>3.75</VForce>
  <IClamp>45</IClamp>
</PowerLevels>
</DCLevels>

```

### 3.4.2 The Block Editor

The block editor is a tool to edit block data in a format suitable for the interactive and test debug environment. Appendix A shows a new Block Editor interface to support a wide view of data (multiple related elements grouped together), whilst retaining the hierarchical nature of the blocks.

## 4 Extensibility of the XML design

A number of special XML blocks, called BlockDefinition blocks, distributed with the software define the syntax of all the other blocks. They are used for several purposes:

- To define whether an element is optional or required.
- To determine the type of each element (string, equation or section) while parsing all other blocks.
- By Block Editor to determine a block's element names and types. For example, when the user wants to add a new element the editor will display a list of those allowed at this point of the syntax.
- By Block Editor to determine "Tool-tip" text for each element.
- By Block editor to determine which elements are references to which other blocks, so that it may "hyperlink" to it.
- By the user, or third-party instrument supplier, when they want to add a new block of their own.

The XML schema file is automatically generated from the BlockDefinition blocks. Below is an example for the definition of the functional test block::

```

<BlockDefinition name="Test">
  <Tool>BlockEditor</Tool>
  <Brief>Test settings.</Brief>
  <Attribute>name</Attribute>
  <String name="Exec">
    <Brief>The fully qualified Java Test template class name</Brief>
  <Occurrence>Required</Occurrence>

```

```

</String>
<Section name="Param">
  <Brief>Definition of a Test Template parameter.</Brief>
  <Attribute>name</Attribute>
  <String name="Type">
    <Brief>The parameter type, e.g. "String" or "DCLevels"</Brief>
    <Enum>DCLevels|DCTestMeasure|DCTestSequence|NameMaps|PatternBurst|Timing|Integer|Double|String</Enum>
  </String>
  <String name="Value">
    <Brief>Value to assign to this parameter.</Brief>
  <Occurrence>Required</Occurrence>
</String>
</Section>
</BlockDefinition>

```

The following is a description of the meaning of some of the possible elements of the block:

- **BlockDefinition**  
This element's attribute is the tag of the block being defined.
- **Tool**

The name of the tool that this block is edited with.

- **Brief**  
A brief description of the element. Block editor displays this as a Tool-tip.

- **Attribute**  
The attribute tag for the block or Section element. Unit Defines the engineering units for an equation expression.

- **String, Equation, Section**  
Defines the type of the element whose tag is specified by the attribute. E.g. <Equation name="VIH">

- **Reference**  
Defines that this element is a reference to a block, and its type. E.g. <Reference>SignalGroups</Reference>

- **Occurrence**  
Takes one or more of the following, with white-space separation:

Required - at least one element is required (elements are optional by default).

Repeatable - the element can be repeated regardless of its type or attribute (by default a String or Equation element can not be repeated, but a Section element can provided that each instance has a different attribute).

Unfixed - this element, together with adjacent "Unfixed" elements, can occur in any order. (By default, element order is fixed.)

- Enum

Explicitly defines the values that a “String” element can take.

The String, Equation and Section sections may themselves contain Brief, Reference and/or Occurrence elements. Section may also contain Attribute, String, Equation and Section elements; sections may be nested indefinitely.

The standard BlockDefinition blocks live in a single file called block\_definitions.xml in the software installation. When the system starts up, just before the test program XML files are loaded, it follows a defined search path to locate and load all block\_definitions.xml files. This allows users and third party developers to define new blocks or add new elements to existing blocks but not to redefine system blocks.

## 5 Conclusion

This paper described the implementation of test data using XML, a license-free, platform-independent and well-supported language, and implementation of test methods using Java, an open, standards-based and platform independent interpretive language in a next generation ATE platform. The advantages of this environment in extensibility using the BlockDefinition block and productivity and reliability through the use of the well adopted standard XML and the open JAVA architecture are obvious. The paper also has provided data comparing performance of Java and C++. The reader is referred to more published material on the web on this subject [6]. When this software architecture proliferates in the ATE domain quite a number of third party software and instrument developers will make major contributions. At that time the impact to the performance of existing test program with respect to test time and load time can be analyzed as a future topic.

## 6 Acknowledgements

The authors would like to thank Colm Gavin and Bill Fritzsche for their clarifications on block structure related questions.

## References

- [1] S.Concina & Neil Richardson, Workstation Driven E-Beam Prober, ITC 1987
- [2] Art Downey, Industry graphics standards & ATE windowing software, ITC 1991
- [3] Don Organ, enVision: the inside story, ITC 1990
- [4] <http://www.w3.org/XML/> World Wide Web Consortium (W3C)
- [5] “IEEE Standard Test Interface Language (STIL): 1450-1999”
- [6] <http://www.onlineessays.com/essays/tech/tech13.php>

## Appendix A: XSchema DCLevels

```

<xs:element name="DCLevels">
  <xs:complexType>
    <!--Any of the following can occur 0
or more times-->
    <xs:sequence>
      <!--Optional single Signal-
Groups block name reference-->
      <xs:element name="Signal-
GroupsRef"
        minOccurs="0" maxOccurs="unbounded"/>
      <!-- ASAP's focus_cal (FX-
specific) stuff no longer needed -->
      <xs:element name="Calibrate"
type="CalibrateType"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="PinLev-
els" type="PinlevelType"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="Power-
Levels" type="PowerlevelType"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="USRSlev-
els" type="USRSlevelType"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="REFlev-
els" type="REFlevelType"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name"
type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="CalibrateType">
<!--Sub-block of DCLevels-->
  <xs:sequence>
    <xs:element name="dut_cap"
type="xs:string" minOccurs="0"/>
    <xs:element
name="levels_focus_cal" type="xs:bool-
ean" minOccurs="0"/>
    <xs:element name="VIH"
type="xs:string" minOccurs="0"/>
    <xs:element name="VIL"
type="xs:string" minOccurs="0"/>
  </xs:sequence>

```

```

    <xs:attribute name="sigref"
type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="PinlevelType">
<!--Sub-block of DCLevels-->
  <!-- ASAP's comp_offset (GX-specific)
no longer needed -->
  <xs:sequence>
    <xs:element name="VIH"
type="xs:string" minOccurs="0"/>
    <xs:element name="VIL"
type="xs:string" minOccurs="0"/>
    <xs:element name="ForceHi"
type="xs:string" minOccurs="0"/>
    <xs:element name="ForceLo"
type="xs:string" minOccurs="0"/>
    <xs:element name="InitHi"
type="xs:string" minOccurs="0"/>
    <xs:element name="InitLo"
type="xs:string" minOccurs="0"/>
    <xs:element name="VOH"
type="xs:string" minOccurs="0"/>
    <xs:element name="VOL"
type="xs:string" minOccurs="0"/>
    <xs:element name="IOH"
type="xs:string" minOccurs="0"/>
    <xs:element name="IOL"
type="xs:string" minOccurs="0"/>
    <xs:element name="LoadVRef"
type="xs:string" minOccurs="0"/>
    <xs:element name="ClampHi"
type="xs:string" minOccurs="0"/>
    <xs:element name="ClampLo"
type="xs:string" minOccurs="0"/>
    <xs:element name="TermVRef"
type="xs:string" minOccurs="0"/>
    <xs:element name="VForce"
type="xs:string" minOccurs="0"/>
    <xs:element name="IClamp"
type="xs:string" minOccurs="0"/>
    <xs:element name="IForce"
type="xs:string" minOccurs="0"/>
    <xs:element name="VClamp"
type="xs:string" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="sigref"
type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Powerlevel-
Type"> <!--Sub-block of DCLevels-->
  <xs:sequence>

```

```

    <xs:element name="VForce"
type="xs:string" minOccurs="0"/>
    <xs:element
name="VForceRange" type="xs:string" mi-
nOccurs="0"/>
    <xs:element name="IClamp"
type="xs:string" minOccurs="0"/>
    <xs:element name="v_bump"
type="xs:string" minOccurs="0"/>
    <!-- ASAP's v_range and delay not
needed on initial DPSs -->
  </xs:sequence>
  <xs:attribute name="sigref"
type="xs:string" use="required"/>
</xs:complexType>

<!-- TBD: -->
<xs:complexType name="USRSlevel-
Type"> <!--Sub-block of DCLevels-->
</xs:complexType>
<xs:complexType name="REFlevelType">
<!--Sub-block of DCLevels-->
</xs:complexType>

```

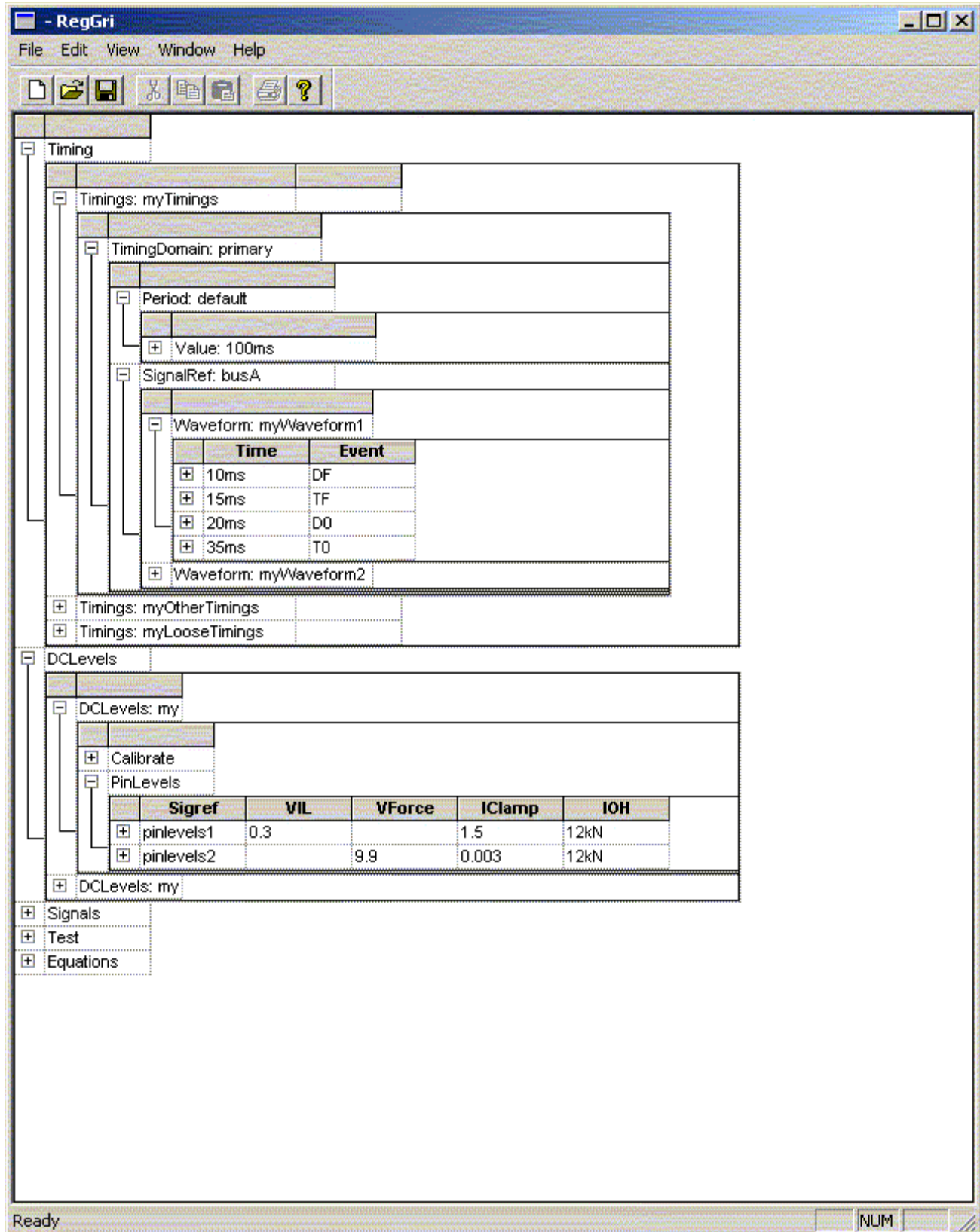


Figure 4. Block Editor For XML Blocks