# STATUS OF THE SpecTcl DATA ANALYSIS PACKAGE

Ron Fox, Chase Bolen, Jeremy Rickard

## 1. Introduction

We have recently developed a powerful, extensible data analysis package. This package, Called SpecTcl is now in use at the NSCL, and several other institutions. Leveraging on the Xamine [1] display server and Tcl/Tk [2][3] scripting, SpecTcl offers an extensible package with a shallow learning curve.

Subsequent sections of this article describe:

- Project Goals, and Development Methodology.
- Feature summary.
- Major components of the package.
- SpecTcl from the user's point of view.

## 2. Project Goals and Methodology

The SpecTcl project is a reaction against the tendency for nuclear physics data analysis packages to be complex works which require a large investment of effort on the part of the user before simple things can be done. Epitomizing these efforts are Paw, Root, and to a lesser extent the Java Analysis Studio.

Paw has a large, complex and inconsistent command language to master as well. Root[4] takes, the approach of providing a class library which the user must warp into a program. By contrast the goals of SpecTcl are:

- Make Simple things easy to do, and complex things possible as well.
- Provide a program which can be modified at well defined plug in points.
- Provide a set of classes which could be used in analysis packages other than SpecTcl.
- Provide no fixed size limits.
- Provide a logical command language which supports extensibility either through scripting or through the addition of commands; and make this extensibility easily accessed.
- Analyze data from any of several common data sources including:

  **Files** Event files.

  **Tapes** ANSI labelled tapes.

  **Online** Online data sources.
- Make SpecTcl easy to adapt to other laboratories with other data acquisition and analysis systems in place.

SpecTcl should be thought of as an Application framework. In program libraries, the main flow of control is managed by the user program. Calls are made to functions in the library or class library to manage the control flow. By contrast, in application frameworks, control flow is managed by the class library. The user creates a working program by subclassing components of the framework, and

substituting for base functionlity. User code typically is written to respond to particular events managed by the program. User code may also sensitize the framework to additional external events and provide handlers for them.

The application framework approach, properly managed, minimizes the amount of code the user must be aware of. In SpecTcl, for example, in most cases it is enough to provide a subclass of the CEventUnpacker class, implementing a single member function which "flattens" input events into an array-like object of type CEvent (an n-tuple if you will).

The application framework of SpecTcl, also has other well defined points at which other sorts of extensions can be made including:

- Pre and post processing the stream of CEvent objects produced by the CAnalyzer object.
- Extending the command set of SpecTcl
- Providing specialized types of spectra and gates.
- Accepting buffer structures which are not NSCL in shape.

A major goal of SpecTcl was to avoid fixed size limits which have been a hallmark of Fortran based analysis codes in NSCL's past. We wanted to do so in a robust manner. Most of SpecTcl's data structures are therefore built on top of or derived from or contain items from the C++ Standard Template Library (STL)[5].

STL includes several useful dynamically sized container classes. The classes most commonly used in SpecTcl include:

**vector** A dynamically resized array (in fact CEvent wraps vector).

**map** A keyed lookup system based on trees. SpecTcl makes extensive use of keyed lookup directories. These are invariably implemented on top of maps.

STL provides generic algorithms which support searching, sorting, and iteration amongst its containers. STL has usually wrapped to permit substitution in the event some other more suitable class library becomes available.

As the name implies, SpecTcl's command language is an extension of Osterhout's Tcl/Tk. SpecTcl consists of commands which extend the functionality of Tcl/Tk to support the definition of event characteristics, spectra and conditions on spectra. Tcl/Tk itself provides powerful scripting support which serve as the underpinnings of SpecTcl's dynamically extensible nature. Tk provides support for scriptable Graphical User Interfaces (GUIs) allowing the user to extend SpecTcl's native GUI to support their own required features and extensions. Commands in SpecTcl provide "introspective" access to SpecTcl allowing analysis to be performed at the Tcl level. Class library wrappers for Tcl allow users to easily register new commands with the Tcl interpreter and hence to further extend SpecTcl's base command language with application specific commands. This process is easy enough that one of the first users of SpecTcl made use of this facility successfully.

Throughout the development of the SpecTcl package, we have used an iterative development methodology. In this methodology, features are incrementally developed and released to the users. The users can then interact with the program at its current implementation level both to get real work done and to provide feedback. Each development cycle incorporates both new features and improvments of existing features based on user feedback from previous cycles.

The key to an iterative development cycle is to try to keep the cycles small and well defined. In this way, the users see a continuous release of new features and improvements in the implementation

quality of older features. Iterative development requires very careful attention to program design as the development team is called upon to modify or re-design the implementation of features in response to feedback.

## 3. Major components of the package

SpecTcl is implemented as a C++ application framework. The framework breaks down into several components. These include:

**TCL** A class library is provided to wrap Tcl/Tk. This class library makes it easy to register new commands, as well as to group commands into interelated packages which share common support services. The user interacts with SpecTcl either through a TkCon console window, typing Tcl/Tk commands, or via a GUI which, in keeping with Tk's philosophy can be extended by the user.

**Data Sources** Data sources are modelled as descendents of a CFile object. CFile's can be checked for available data as well as read. Current "normal" data sources include disk files and files on ANSI labelled tapes. Two rather interesting event sources include a configurable internal test data source and a pipe data source. The test data source produces fixed length events containing random data. The pipe data source reads data from a UNIX pipe attached to a command specified when the data source is set up. Pipe data sources were intended to connect SpecTcl with online data acquisition systems, however they have also been used to facilitate batch processing of multiple run files as well as processing runs from event data files compressed with e.g. gzip.

**Event Analyzer** The analyzer component accepts buffers of data from a Data Source and picks it apart into events. This unpacking consists of two components. A component aware of buffer internal structuring and a component which is aware of the structure of events in physics data buffers. These decode the buffer and "flatten" events into a CEvent array class.

**Sorting** CHistogrammer is actually a special case of a data sink attached to the Event Analyzer. CEventLists (vectors of CEvent objects) are produced and sent to data sinks. CHistogrammer understands how to sort data into histograms. It maintains lists of parameter definitions, histogram definitions and condition definitions which are referenced during the sorting process. Sophisticated user programmers could supply a non-sorting event sink which might do other things, like produce data summary tapes.

**Displayer** This set of classes serves as an interface between the histogrammer and the display services of some visualization program. Currently Xamine is the only displayer supported. In the future, to support non-interactive production mode analysis, we will support a "null" displayer as well.

**Glue** A set of classes whose purpose is to string together the components of SpecTcl into an application framework.

Wherever possible, SpecTcl classes and components have pushed policy decisions into the "Glue" components. This promotes a higher degree of re-usability. With usage policy decisions in each component, the components would tend to become very tightly coupled together making them hard to re-use outside of the SpecTcl program. An example of this sort of tight binding is Root's tight binding of the display mechanism to its CHistogram class subtree. SpecTcl histograms can be instantiated independent of a display allowing the program to be run without displays or with other display mechanisms.

## 4. SpecTcl from the user's point of view

SpecTcl users must:

- Write C++ code to "flatten" their event data into CEvent n-tuples.
- Interact with the program to get it to produce the desired set of histograms.

The web pages at http://www.nscl.msu.edu/ fox/SpecTcl/index.htm provides a current online user and reference guide to SpecTcl. These html pages are provided with SpecTcl distributions and the current GUI includes a HELP button which points a netscape web browser at the these pages. In this section, we will summarize the sorts of things which must be done to configure SpecTcl to analyze a data set. We will also show interactions the user might have with SpecTcl during an analysis session.

4.1. "Flattening" Events

SpecTcl's Event Analyzer subsystem takes a stream of buffers from a CFile derived object. The output of the Analyzer is a stream of CEvent "flattened" events. CEvent is a wrapped STL vectors of parameters. The Analyzer sends CEventList objects, which are arrays of CEvent objects to an event sink object. In the case of SpecTcl, the Event sink is by default a histogrammer.

The Analyzer unpacks events making use of two objects which are attached to it at initialization time. The first item, subclassed from CBufferDecoder understands the global structure of data buffers. It takes a raw block of data from the event source and back calls appropriate member functions in the Analyzer as appropriate to the data contained by the buffer. Unmodified, SpecTcl, connects a CNSCLBufferDecoder object to the analyzer class. CNSCLBufferDecoder objects understand the gross structure of NSCL raw data buffers.

When a BufferDecoder object encounters event data it calls the Analyzer's OnPhysics member. This function iterates through the data passing events to a CUnpacker's operator() (function call operator) virtual member function. For NSCL data, this is the only function which the user will have to write.

The Unpacker's function call operator receives a pointer to the event, a reference to a CEvent object into which to attempt to flatten the raw data and references to both the analyzer and the buffer decoder. The CEvent object is initialized so that all elements of the event are undefined. The unpacker has the following three options:

- Unpack the event into the CEvent object (flatten successfully) and return the number of bytes decoded so that the Analyzer knows where the next event is.
- Discover a recoverable event format error (e.g. a failure of the readout software). Recoverable event format errors are thrown as exceptions caught by the analyzer. The exception results in an error message. The size of the event determined by the unpacker is part of the exception object and is used to continue processing the same buffer of data with the next event.
- Discover an unrecoverable event format error. Unrecoverable event format errors are thrown as exceptions caught by the analyzer. These exceptions result in an error message as well as abandonment of the current buffer.

Flattening functions can naturally produce *pseudo parameters* which are referred to as *compiled psuedos*.

Once the user has written an unpacking function to flatten the events, make is used to build a tailored version of SpecTcl for the user.

4.2. Sample Interactions SpecTcl commands and operations are all performed by executing Tcl command extensions and Tcl scripts. This has several consequences:

- There is no separate configuration file format to setup SpecTcl's initial state, Tcl scripts can just be sourced in by the user. SpecTcl also automatically reads a startup script which may contain this setup information.
- Commands and operations can easily be bound to extensions to the user interface written in Tk.
- The rich set of control structures offered by SpecTcl makes it easy to make complex and repetitive definitions.

A set of Tcl class wrappers makes it easy for users to add their own commands to SpecTcl, access Tcl variables and arrays, or bind variables to C/C++ variables.

The remaining parts of this subsection describe:

- How parameters and histograms are defined.
- How conditions are defined and applied to conditionalize spectrum increments.
- How scripted pseudos can be dynamically defined.
- Introspective features allowing complex scripted functions.

### 4.2.1 Defining parameters and histograms

This section describes the *parameter* command and the *spectrum* command. These two commands provide SpecTcl's mechanism for naming parameter positions in the flattened CEvent objects produced by the user's unpacker, and describing a set of spectra to be produced from these parameters. Future work in SpecTcl will focus on enriching the options supported by the spectrum command as well as enlarging the set of spectrum types supported. We also intend to provide support for imposing a hierarchical naming system on the parameter space.

The parameter command gives a parameter name and characteristics to a slot in the CEvent array. The defining form of the command is:

```
parameter name slot bits
```

Where:

**name** is a descriptive name which will be used to refer to the parameter.
**slot** is a number indicating which slot in the CEvent "array" the parameter will be flattened.
**bits** is a number indicating the number of bits of resolution the parameter uses.

Once a parameters have been defined, histograms can be defined on the parameters. SpecTcl allows the definition of an arbitrary number of histograms. SpecTcl and Xamine communicate via a shared memory region with a size which is fixed at startup time. SpecTcl commands allow spectra to be bound or unbound from this display memory. It is therefore possible to use Xamine to view some subset of a larger set of spectra and then switch to some other subset. Note that binding and unbinding involves a physical relocation of the channel data and modification of the attributes of the underlying CSpectrum derived object so that no double incrementing will ever take place.

Histograms are defined using the spectrum command. The defining form of that command is:

```
spectrum name type paramlist reslist [dtype]
```

Where:

**name** is a name which will be used to refer to the spectrum.

**type** is the type of spectrum being created.

**paramlist** is a properly formatted Tcl list containing the set of parameters needed to increment the histogram.

**reslist** is a properly formatted Tcl list containing resolution specifications for the spectrum. Currently a resolution specification is just the number of bits of resolution used by the spectrum. In the future this will be upgraded to support resolutions which describe ranges within the parameter spaces.

**dtype** Is an optional parameter which if present can be one of $b$ $w$ $l$ indicating that the channels should be held inside a byte (8bits), word (16bits) or longword (32 bits) respectively.

SpecTcl already supports a rich set of spectrum types. These include:

**1** Ordinary 1-d histogram.

**2** Ordinary 2-d histogram.

**b** 1-d histogram incremented once for each bit set in the parameter. The channels of the histogram represent bit positions.

**s** Summary histogram. This is a special 2-d histogram intended to summarize the state of many similar elements of segmented detector systems. An arbitrary number of parameters may be specified, along with a y axis resolution. The X channel represents parameters with valid data, while the Y channels are parameter values.

### 4.2.2 Defining and applying conditions

Histograms may be gated on a single arbitrarily complex gate condition. Gates are evaluated recursively, and results of evaluations are cached within each event. Gates can be defined by Tcl commands. In addition, SpecTcl interacts with Xamine to accept primitive gates (cuts, bands and contours) entered graphically by the user.

The defining form of the gate command is:

```
gate name type description
```

Where:

**name** Is the name which will be used to identify the gate.

**type** Is the type of the gate being created.

**description** is a gate type dependent description of the gate.

SpecTcl supports many different types of gates. Primitive gates in general represent simple decisions, while compound gates can be composed of both primitive and other compound gates to build up arbitrarily complex gates. The gate description is a properly formatted Tcl list with contents that depend on the gate type.

Primitive gates supported by SpecTcl, along with their descriptions include:

**T** A gate which is always true. By default when a spectrum is created, it is gated by a True gate. This is an instance of the "null object pattern" *reference to pattern in PLOP* which simplifies the handling of ungated spectra. The True gate has no description.

**F** A gate which is always false. This gate can be applied to effectively disable the increment of a spectrum. The False gate has no description.

**s** A "Slice" gate representing a slice in parameter space. Slice gates represent cuts accepted graphically in Xamine. Slice gates are also entered back into Xamine so that they display on all 1-d histograms of the associated parameter. The description of a slice gate contains two elements. The first is the name of the parameter on which the slice is defined. The second is a sub-list containing the lower and upper limits of the cut in parameter coordinates.

**b** A "band" gate representing all of the space below a polyline. Bands represent Band gates entered through Xamine. If the band does not extend to the edges of the spectrum, its value is assumed to drop to zero excluding areas to the left and right of the band. The description of a band consists of a Tcl list whose elements are: The X parameter name, the Y parameter name, and a list containing elements which themselves are lists containing the X and Y coordinates of the band points. The band is represented internally as an upper/lower limit pair lookup table indexed by the X parameter value. Band gates are entered into all 2-d spectra in Xamine which are defined on the pair of parameters making up the gate.

**c** A "Contour" gate representing all of the space inside a polygon. Contours are used to represent contour gates entered graphically by the user with Xamine. The interior of complex figures is determined using the even crossing rule. The *reference graphs* algorithm, is used to create a bitmap of coordinate pairs made by the gate. This bitmap is circumscribed by a bounding rectangle in order to reduce storage requirements and improve evaluation speed in the typical case (contour is a probabilistically small region of parameter space). The description of a contour is identical to that of a band.

Compound gates are made up of other previously defined gates. Gates which make up compound gates can be either primitive gates or other compound gates allowing arbitrarily complex logical expressions to be incrementally constructed. In the future we will implement an expression evaluator to develop complex gates out of arbitrary logical expressions.

SpecTcl supports the following compound gates:

**-** A Not gate. The description consists of a single gate. The Not gate returns the inverse of the value of this gate.

***** An And gate. The description of an and gate is a list of gate names. The gate is made only if all constituents are made. Short circuit evaluation is used to avoid evaluating gates which are not required to determine the gate value.

**+** An Or gate. The description of an or gate is a list of gate names. The Or gate is made if any of the constituents is true. Short circuit evaluation is used to avoid evaluating gates which are not required to determine the gate value.

**c2band** A contour created from two bands. The "upper" band is joined at endpoints to the "lower" band to form a countour. The gate is made by points inside this contour. The description consists of the two constitutent gates which must be bands.

Figure 1 shows how complex gates can be built up.

Once created gates can be applied to a spectrum using the *apply* command. The form of this command is:

```
apply gate spec1 [spec2...]
```

```
gate E1cut  s {E1 {100 200}}             ;# an energy cut
gate E2cut  s {E2 {300 576}}             ;# another energy cut.
gate Alphas1 c {dE1 E1 {.......}}        ;# A contour (points omitted)
gate protons2 c {dE2 E2 {...}}
gate Notproton2 - protons2               ;# No protons allowed
gate Pidrequire * {Notproton2 Alphas1}   ;# Alphas in 1 and not proton in 2
gate Final   * {E1cut E2cut Pidrequire}  ;# Makes energy and PID requirements.
```

Figure 1: Sample annotated gate script

#### 4.2.3 Scripting in event processing

Pseudo parameters can be created in the users's flattening code. Sometimes, however it is nice to be able to create ad-hoc calculated parameters. SpecTcl supports scripted pseudos, allowing the user to effectively create sets of Tcl procedures which are called for each event to compute these parameters.

This feature makes use of the fact that:

- Parameters can be defined on "unused" slots in the CEvent array.
- Compiled Tcl extensions can create and invoke Tcl scripts.

The SpecTcl pseudo command is used to define a scripted pseudo. The form of this command is:

```
pseudo name usedparams {
   procedurebody
...
}
```

Where:

**name** Is the name of a parameter which has been defined and is not produced by the flattener, this is the parameter which will be produced by the scripted pseudo.

**usedparams** is a list of parameter names which are required by the pseudo to produce a result. These can be simple parameters or previously defined pseudos.

**procedurebody** is a Tcl procedure body which returns an integer value that is the value of the pseudo parameter. Parameters in the used parameter list can be referenced using standard Tcl substition rules.

SpecTcl uses this description to build a procedure parameterized by the parameters in the usedparams list. When an event is processed, a call to this procedure is generated and parameterized by the actual values of these parameters.

#### 4.2.4 Introspective capabilities

SpecTcl and Tcl's power is that it provides a simple language in which very complex operations can be represented. It provides a consistent base language on top of which application specific extensions can be layered. The extent to which programs like SpecTcl can be extended to perform tasks not originally envisioned depends on how much of the internal data is exposed to the script level and in what way it is exposed. This ability, of a Tcl extension to look into its own internals via its own commands is called introspection.

All of SpecTcl's defining commands also have inquiry forms which allow user scripts to determine which objects have been created. The output of these commands, while human readable, are also intended to be processed by Tcl scripts. An additional *chan* command allows access to the channels of spectra.

It is possible, for example to write scripts to do spectrum background determination, spectrum addition peak fitting and other sorts of analysis. We anticipate that a library of these scripts will be built by both the NSCL computer staff and the users of SpecTcl.

## 5. Conclusions

User experiences with SpecTcl have been largely positive. It is relatively easy to start simple as well as complex data analysis tasks. People have extended SpecTcl with C++ add-ons as well as scripts for their own applications. Several external users of SpecTcl have also emerged and the future for this development project looks bright.

## References


1. *The Xamine Online/Offline Display Program* R. Fox et al. IEEE Trans. on Nucl. Sci. **NS-43** No. 1 55-60
2. Tcl and the Tk toolkit John. K. Ousterhout Addison-Wesley May 1994
3. Practical Programming in Tcl and TK Brent B. Welch Prentice Hall January 2000
4. http://root.cern.ch
5. Stl Tutorial & Reference Guide: C++ Programming with the Standard Template Library D.R. Musser, A Saini and A. Stepanov Addison-Wesley March 1996