Paradyn Parallel Performance Tools

VisiLib Programmer's Guide

Release 4.1
April 2004

Paradyn Project
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
paradyn@cs.wisc.edu

Table Of Contents

List of Figures

1 PRELIMINARIES

This guide documents *VisiLib* - a library and remote procedure call interface for accessing Paradyn performance data in real-time. VisiLib provides an open interface to Paradyn data, and allows a programmer to build external visualization processes (*visis*). All performance visualizations in Paradyn are implemented as visis.

VisiLib handles all the messy details of communicating with Paradyn, and of receiving and storing Paradyn performance data. The visi programmer can access performance data by calling VisiLib routines. All menu control and selection of the data to be displayed is handled in Paradyn itself.

1.1 Document revision history

- **v4.0**: minor modifications
- **v3.3**: minor modifications
- **v3.2**: minor modifications
- **v3.1**: minor modifications
- **v3.0**: minor modifications
- **v2.1**: minor corrections
- **v2.0**: initial release.

1.2 Overview

Figure 1 shows the visualization interface in relation to the Paradyn process, external visualization processes, and application processes. It also shows tclVisi, the Tcl interface to VisiLib.

Visualization processes are started by the Paradyn process; once initialization has been done between the two, requests for performance data can be made by the visualization process. By making requests for data, visualization processes may cause data flow to be enabled or disabled from the application processes. Once enabled, data values are sent from Paradyn to visualizations. Visualization processes register callbacks on Paradyn events. For example, if a visualization wants to display performance data as it arrives from Paradyn, it would register a callback routine that would be called whenever new data arrived from the Paradyn process.

The following sections describe the details of the VisiLib interface. Section 2 gives a detailed explanation of VisiLib interface functions. Section 2.3 describes the Tcl interface to VisiLib that can be used to add Tcl/Tk[1] visualizations to Paradyn. Section 4 describes how to add a visualization definition to a PCL (Paradyn Configuration Language) file, and Section 5 contains two examples.

1. “Tcl and the Tk Toolkit”, by John Ousterhout, published by Addison-Wesley.

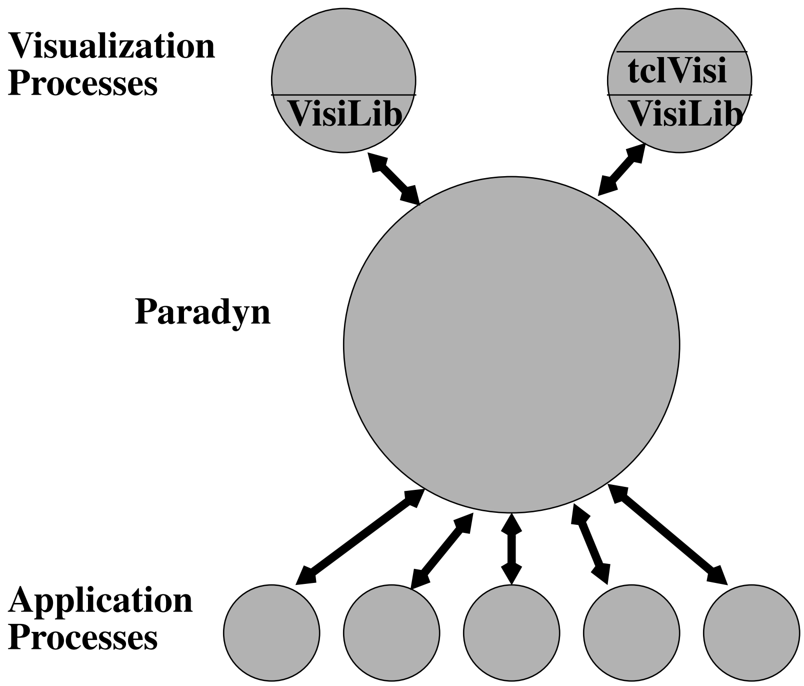

Figure 1: Visualization interface

2 VISI INTERFACE

The following sections give a detailed description of VisiLib interface functions and the methods for adding a visualization to Paradyn. Section 2.1 describes the types defined by VisiLib, Section 2.2 describes the VisiLib interface, and Section 2.3 describes tclVisi (a tcl interface to VisiLib).

2.1 Types

2.1.1 Metrics and Resources

Visualizations request data by specifying a list of metric and resource combinations. A metric is a value that represents some aspect of a program's performance. Examples of metrics are CPU time or procedure calls per second. A resource is a specification of a part of a program's execution expressed as a collection of program objects. Example objects are synchronization objects (e.g., semaphores or message tags), code objects (e.g., modules or procedures), and process objects. As an example, a visualization could request data collection for the metric "CPU Utilization" and the resource "`/Code/foo.c/Process/{i}`". In this example, CPU utilization would be measured for process `{i}` when it is actively executing code in module foo.c.

Metric and Resource combinations are initially selected from menus created by Paradyn. Once a set of metrics and resources has been selected, a visualization process can obtain information about this set by making calls to VisiLib functions. These functions are discussed in Section 2.2.3.1 and in Section 2.2.3.2.

2.1.2 Histogram

Histograms are used by VisiLib to store performance data that is sent by Paradyn. The histogram data structure is a fixed length array. Each element (bucket) in the array represents a time interval and stores the value of a metric over that interval. The granularity of the performance data stored in a histogram is determined by the bucket width (time interval), and by the number of buckets in the histogram. When the last histogram bucket is filled with a data value, the histogram buckets are folded. Folding consists of doubling the bucket width and combining the values of two adjacent buckets into one bucket of the new interval size. The resulting histogram has twice the previous bucket width, leaving half the buckets empty. Since folding is an event that may be of interest to a visualization, VisiLib provides a mechanism for notifying the visualization when a fold has occurred. This mechanism is discussed in Section 2.2.2.

2.1.3 Data Grid

The DataGrid is a C++ class that provides the visualization application's interface to performance data. The DataGrid can be thought of as a two dimensional array that is indexed by metricId and resourceId. Each *(i,j)* element in the array is a histogram of performance data. The VisiLib library contains functions for accessing Paradyn performance data and meta-data that is stored in the DataGrid.

2.1.4 Phases

Phases in Paradyn are contiguous time intervals within an application's execution; there is exactly one current phase in the system at any time. Visualization processes can be defined for either the global phase or the current phase (this is specified through a menu option when the visualization is started from Paradyn). Visualizations that are defined for the global phase, will receive performance data that spans the entire run time of the application. Current phase visualizations display performance data that spans the time interval from the current phase's start time to the end of the current phase. Typically, data for the global phase is at a coarser granularity than current phase data.

When a new phase is defined in the system, the current phase ends and all data collection for the current phase stops. Visualizations defined for the current phase will no longer receive performance data from Paradyn. Global phase visualizations will continue to receive data for the entire run of the application or until the visualization requests that data collection stop.

Since phase start and end events may be of interest to visualizations, VisiLib provides a mechanism for notifying a visualization when these events occur. This mechanism is discussed further in Section 2.2.2. VisiLib also provides functions for accessing information about phases in the system. Phase information consists of a string name, a unique handle, a start time, and an end time for each phase. These functions are discussed in Section 2.2.3.6.

2.1.5 visi_timeType

VisiLib defines the type `visi_timeType` for representing Paradyn time valued data. Histogram bucketwidth, phase start and phase end times are represented by `visi_timeType` values. The definition for this type is in the header file `visiTypes.h`, and is currently defined as a double.

2.1.6 visi_sampleType

VisiLib defines the type `visi_sampleType` for representing Paradyn performance data. Histogram bucket values are represented by `visi_sampleType` values. Currently `visi_sampleType` is defined as a float.

2.1.7 visi_TraceData

VisiLib defines the type visi_TraceData for representing Paradyn trace data. Currently, visi_TraceData is defined as a class, which has metricIndex of integer type, resourceIndex of integer type, and dataRecord of byteArray-pointer type as its members. ByteArray is a class that contains a pointer to a byte array and its length. The use of trace data is discussed in Section 2.4.

2.2 VisiLib interface functions

This section describes the VisiLib interface functions. We begin by discussing how to use the VisiLib interface functions directly; then we discuss how to use tclVisi the Tcl interface to VisiLib. Within each section we explain initializing VisiLib, using Paradyn events, accessing Paradyn performance data and meta-data, and making calls to Paradyn.

2.2.1 Initialization

```
PDSOCKET visi_Init()
```

All visualizations should call the `visi_Init` VisiLib function before entering their main loops. This routine takes care of setting up the connection between the Paradyn and visualization processes. Once this routine is called, the visualization process can call any other VisiLib routines. The returned PDSOCKET represents the connection to Paradyn, and the application should register this PDSOCKET as a potential source of input. When input is seen on the PDSOCKET, the application should call the `visi_callback` VisiLib function to process the input.

If the call fails, the function returns PDSOCKET_ERROR.

2.2.2 Paradyn events

```
int visi_RegistrationCallback(msgTag event, int (*callBack)(int))
```

Visualizations can register callback routines on any Paradyn event by calling the `visi_RegistrationCallback` interface function with an event type and a callback routine as arguments. A Paradyn event corresponds to either data or some type of information that has been sent by Paradyn and has been received by the visualization process. The different event types are listed in Figure 2. A callback is a function that is supplied by the visualization writer that will be called by a VisiLib routine that handles the associated Paradyn event.

The `callBack` argument to `visi_RegistrationCallback` is the name of a function provided by the visualization writer. The callback routine is called with an `int` argument that is used by the DATAVALUES event to indicate the last histogram bucket number with new performance data, or to specify the phase identifier for PHASESTART and PHASEEND events. For all other events this parameter is not used. When an event occurs, the callback routine can call DataGrid method functions to get data corresponding to the event. For example, to add a callback routine for the DATAVALUES event, a visualization writer would make the following call:

```
int ok = visi_RegistrationCallback(DATAVALUES, myDataHandler);
```

In this example, the callback routine `myDataHandler` would be called by a VisiLib routine whenever new performance data is received from Paradyn. The VisiLib routines would handle storing the performance data in the DataGrid, and the `myDataHandler` callback could then use VisiLib functions to access the new performance data and update its displays with the new data. The functions for accessing performance data are discussed in Section 2.2.3.

DATAVALUES	New data values have arrived from Paradyn and have been added to the DataGrid. They can be accessed through DataGrid method functions
INVALIDMETRICSRESOURCES	A previously valid metric/resource combination has become invalid.
ADDMETRICSRESOURCES	A new set of metrics and resources have been added to the DataGrid
PHASESTART	A new phase has been defined.
PHASEEND	The current phase has ended.
PHASEDATA	A set of phase information has been added to the DataGrid.
FOLD	The DataGrid histogram structures have folded.
PARADYNEXITED	The Paradyn process has exited. If there is a callback registered on this event it will be called by VisiLib, otherwise VisiLib kills the visualization process.
TRACEDATAVALUES	A new trace data record has arrived from Paradyn and has been stored in the visi_TraceData instance. Since the same instance will be replaced by the next-arriving data record, the record should be copied if it is needed after the callback returns.

Figure 2: Paradyn event types

Similarly, for trace data, a callback routine can be registered on the TRACEDATAVALUES event that will be called by a VisiLib routine when the TRACEDATAVALUES event occurs. The

VisiLib routine will handle storing the trace data in a visi_TraceData instance, and the callback routine could then call a VisiLib function to access the new trace data, and perform any analysis on the data. The VisiLib function for accessing trace data is discussed in Section 2.2.3.

Calls to `visi_RegistrationCallback` should be made before the application enters its main loop. A visualization writer need only register callback routines for those events in which the visualization is interested. A call to `visi_RegistrationCallback` will fail, if an invalid event is specified. If a call fails, it will return VISI_ERROR_INT; on success, it will return VISI_OK.

2.2.3 Getting data from the data grid

The DataGrid provides the interface to all Paradyn performance data and meta-data. Visualization writers can access this data by calling VisiLib functions that access the DataGrid. Typically these functions are used in callback routines that have been registered on Paradyn events.

2.2.3.1 Accessing metric data

```
int visi_NumMetrics()
```

The `visi_NumMetrics` function returns an integer value that is the total number of metrics in the DataGrid, typically used as a loop bound when accessing data values from the DataGrid.

```
const char *visi_MetricName(int i)
```

The visi_`MetricName` function takes an integer argument *i* representing the i^{th} metric in the DataGrid and returns the character string representation of the i^{th} metric's name.

```
const char *visi_MetricLabel(int i)
```

The `visi_MetricLabel` method function takes an integer argument *i* representing the i^{th} metric in the DataGrid and returns the character string representation of the i^{th} metric's units label. An example of a units label is "Calls / Second".

```
const char *visi_MetricAveLabel(int i)
```

The `visi_MetricAveLabel` method function takes an integer argument *i* representing the i^{th} metric in the DataGrid and returns the character string representation of the i^{th} metric's average units label This would be used for labeling the metric's units when the average over all the data buckets is being displayed.

```
const char *visi_MetricSumLabel(int i)
```

The `visi_MetricSumLabel` function takes an integer argument *i* representing the i^{th} metric in the DataGrid and returns the character string representation of the i^{th} metric's sum units name. This would be used for labeling the metric's units when the data value being displayed is the sum of all the data buckets for the metric. An example of a sum units label is "Calls".

2.2.3.2 Accessing resource data

```
int visi_NumResources()
```

The `visi_NumResources` function returns an integer value representing the number of resources in the DataGrid, typically used as a loop bound when accessing data values from the DataGrid.

```
const char *visi_ResourceName(int i)
```

The `visi_ResourceName` function takes an integer argument *i* representing the i^{th} resource in the DataGrid and returns the character string representation of the i^{th} resource's name.

2.2.3.3 Accessing performance data values

```
visi_sampleType visi_DataValue(int m,int r,int b)
const visi_sampleType *visi_DataValues(int m,int r)
```

The `visi_DataValue` function retrieves data values from the DataGrid. The first argument (`m`) specifies the metric, the second (`r`) specifies the resource, and the third (`b`) specifies the bucket number of the corresponding piece of data. Data values are stored as floats. DataGrid buckets that do not contain valid performance data have a NaN float value. Visualizations then can test bucket values for valid data before calling routines that display the data. For example, in the section of code below, bucket values are tested for valid data before making a call to a routine that displays the bucket value. A call to the *isnan* routine defined in math.h can be used to determine the validity of the data. For example:

```
if(!isnan(visi_DataValue(m,r,b))) {
      Display_Data(visi_DataValue(m,r,b));
}
```

The `visi_DataValues` function returns a pointer to an array of data values for the specified metric and resource pair. If there are no data values corresponding to metric `m` and resource `r`, then this routine returns a NULL pointer. The number of data values in the array can be obtain by calling the `visi_NumBuckets` VisiLib routine. This routine is discussed in Section 2.2.3.4.

```
visi_sampleType visi_AverageValue(int m,int r)
visi_sampleType visi_SumValue(int m,int r)
```

The `visi_AverageValue` and `visi_SumValue` functions return the average and sum of all the valid data buckets for the performance data associated with metric `m` and resource `r`. These routines return a NaN floating point value if there are no valid data buckets for the metric-resource pair.

2.2.3.4 Accessing information about DataGrid elements

```
int visi_Valid(int m,int r)
int visi_Enabled(int m,int r)
```

Not every pair `(m,r)` of a metric `m` and a resource `r` contains a histogram of Paradyn performace data. Because of this, VisiLib provides functions for testing whether or not an `(m,r)` pair is *vali*d and for testing whether or not an `(m,r)` pair is *enabled*. We define a *valid* pair as one that has an associated histogram in the DataGrid and we define an *enabled* pair as one which is ready to receive data from Paradyn. The `visi_Valid` and `visi_Enabled` functions take two arguments. The first is the index of a metric, and the second is the index of a resource. `visi_Valid` returns a value of 1 if the corresponding DataGrid cell contains a histogram of performance data, and returns a value of 0 otherwise. The `visi_Enabled` method function returns 1 if data collection for the `(m,r)` pair has been enabled, and 0 otherwise. Note that, since being in the enabled state is a prerequisite for validity, some `(m,r)` pairs can be *enabled* but not *valid*.

```
int visi_NumBuckets()
```

The `visi_NumBuckets` function returns the total number of histogram buckets in each histogram in the DataGrid (note: all DataGrid histograms have the same number of buckets). This value is fixed at runtime; it cannot be changed by a visi and is typically used as a for loop bound.

```
visi_timeType visi_BucketWidth()
```

The `visi_BucketWidth` function returns the length of the time interval represented by each histogram bucket. This is typically used for computing axis labeling in time-plots.

```
int visi_FirstValidBucket(int m,int r)
int visi_LastBucketFilled(int m,int r)
```

These functions return the first and last histogram bucket with valid performance data for the histogram in cell `(m,r)`. It is possible for spans of invalid data between these two points to exist.

```
int visi_InvalidSpans(int m,int r)
```

This function returns 1 if there exist invalid spans of data between the first bucket filled and the last bucket filled in the DataGrid cell corresponding to metric `m` and resource `r`. Otherwise, this routine returns a value of 0.

2.2.3.5 DataGrid user data

VisiLib provides routines that allow a visualization writer to associate other data with DataGrid cells. VisiLib stores this data as a void pointer.

```
void *visi_GetUserData(int m,int r)
int visi_SetUserData(int m,int r,void *my_data)
```

The `visi_SetUserData` function takes two integer arguments indicating which metric and resource in the DataGrid to associate the data with. The third argument is a pointer to the data to be added to DataGrid cell `(m,r)`. This routine returns 1 if the data pointed to my `my_data` was successfully added to the DataGrid, otherwise this routine returns 0.

The routine `visi_GetUserData` is used to retrieve user data from a DataGrid cell. It is called with two arguments; the metric number and the resource number of the cell in the DataGrid. This routine returns a NULL pointer if the user data cannot be retrieved from DataGrid cell `(m,r)`.

2.2.3.6 Accessing phase data

A visualization is associated with exactly one phase: either the global phase or the current phase. Visualizations that are associated with the current phase stop receiving performance data from Paradyn when the current phase ends. Global phase visualizations continue to receive performance data for the entire run of the application or until the visualization explicitly requests that data collection stop. VisiLib provides functions for accessing information about the visualization's own phase and about all other phases defined in the system.

```
const char *visi_GetMyPhaseName()
```

The `visi_GetMyPhaseName` method function returns the character string name associated with the phase for which the visualization is associated.

```
visi_timeType visi_GetStartTime()
```

The `visi_GetStartTime` method function returns the start time associated with the phase. This value is useful for labeling time axes. Only the global phase and the first non-global phase start at time zero.

```
int visi_GetMyPhaseHandle()
```

This routine returns the unique phase identifier for the phase in which the visi is defined. If the visi is defined for the global phase, then this routine returns -1, otherwise it returns a non-negative integer value.

```
int visi_NumPhases()
```

The `visi_NumPhases` method function returns an integer value representing the number of phases currently defined in the system. This value can be used as an upper bound when querying the DataGrid about a specific phase.

VisiLib provides functions for accessing information about other phases defined in the system. This is useful for accessing information about phases for which the visualization is not defined: past phases if the visualization is defined for the current phase, and past and current phase info if the visualization is defined for the global phase. The following is a description of the VisiLib phase functions:

```
const char *visi_GetPhaseName(unsigned i)
```

The `visi_GetPhaseName` method function returns a character string representation of the ith phase's name. This is typically used for labeling phases that are displayed by the visualization.

```
visi_timeType visi_GetPhaseStartTime(unsigned i)
```

The `visi_GetPhaseStartTime` function returns a visi_timeType value representing the start time of the ith phase.

```
visi_timeType visi_GetPhaseEndTime(unsigned i)
```

The `visi_GetPhaseEndTime` function returns a visi_timeType value representing the end time of the ith phase. If the phase has not yet ended, this function returns a value of -1.0.

```
int visi_GetPhaseHandle(unsigned i)
```

This routine returns the unique identifier for the ith phase.

2.2.3.7 Accessing trace data values

```
#include "visi/src/visiTypesP.h"
visi_TraceData *visi_TraceDataValues();
```

The visi_TraceDataValues function retrieves a pointer to a visi_TraceData class instance. The class has three members: metricIndex, resourceIndex, and dataRecord. The dataRecord class has getArray and length as its member functions, which return a pointer to a byte array and its length, respectively. For example:

```
char *sp = visi_TraceDataValues()->dataRecord->getArray();
int len = visi_TraceDataValues()->dataRecord->length();
```

2.2.4 Calls to Paradyn

The VisiLib library provides routines to call Paradyn to change the set of performance data that is being sent to the visualization process. These calls are asynchronous, and may eventually result in a Paradyn event arriving from Paradyn. For example, by requesting more data from Paradyn, an ADDMETRICSRESOURCES event may eventually occur indicating that a new set of metrics and resources has been added to the DataGrid and that performance data values for this set will start arriving. The following is a description of these calls:

```
void visi_GetMetsRes(char *metres,int numElements)
```

The `visi_GetMetsRes` routine requests Paradyn to display menus to select new metric and resource combinations to be enabled. When Paradyn receives this request, it will initiate menuing for new metric and resource selections. The first argument is a list of pre-selected metric-resource pairs, and the second argument is the number of pairs in the list. Currently, VisiLib does not support passing a pre-defined list of metric-resource pairs to enable, so a call to this routine with any arguments is equivalent to the following: `visi_GetMetsRes(0,0)`.

```
void visi_StopMetRes(int metricIndex, int resourceIndex)
```

The `visi_StopMetRes` routine requests Paradyn to stop data collection for the metric and resource pair specified by the metricIndex and resourceIndex arguments. These arguments are the indices into the DataGrid corresponding the metric-resource pair for which data collection is to be terminated. When this request is made, the corresponding DataGrid cell is marked invalid; when the paradyn process receives the request, it will disable data collection for this visualization for the specified metric-resource pair.

```
void visi_DefinePhase(char *name, unsigned with_pc, unsigned with_visis)
```

The `visi_DefinePhase` routine makes a request to Paradyn to start a new phase. The first argument is a name that can be specified to be associated with the new phase. Passing in a NULL value will result in Paradyn creating a name for the new phase. If `with_pc` is 1 and the Performance Consultant window is open in Paradyn, then a new Performance Consultant search will begin on this new phase immediately. If `with_pc` is 0, then a new Performance Consultant search will not be started with the new phase. The `with_visis` option is not yet implemented. Invoking this routine will eventually result in a PHASEEND event from Paradyn corresponding to the end of the current phase, and a PHASESTART corresponding to the start of the new phase.

```
void visi_showErrorVisiCallback(const char *msg)
```

The `visi_showErrorVisiCallback` routine makes a request to Paradyn to display the error message specified by the argument `msg`. Paradyn will use it's error message window to display the error message. For more information on Paradyn's error message interface see the ***Paradyn User's Guide***.

2.3 Tcl Interface to VisiLib

We provide a tcl interface for the visi interface, *tclVisi*, that allows programmers to use Paradyn performance data in tcl applications. Figure 3 shows tclVisi in relation to VisiLib, Paradyn, the application processes, and the tcl code for the visualization. TclVisi contains a tcl interpreter which interprets the visualization's tcl code.

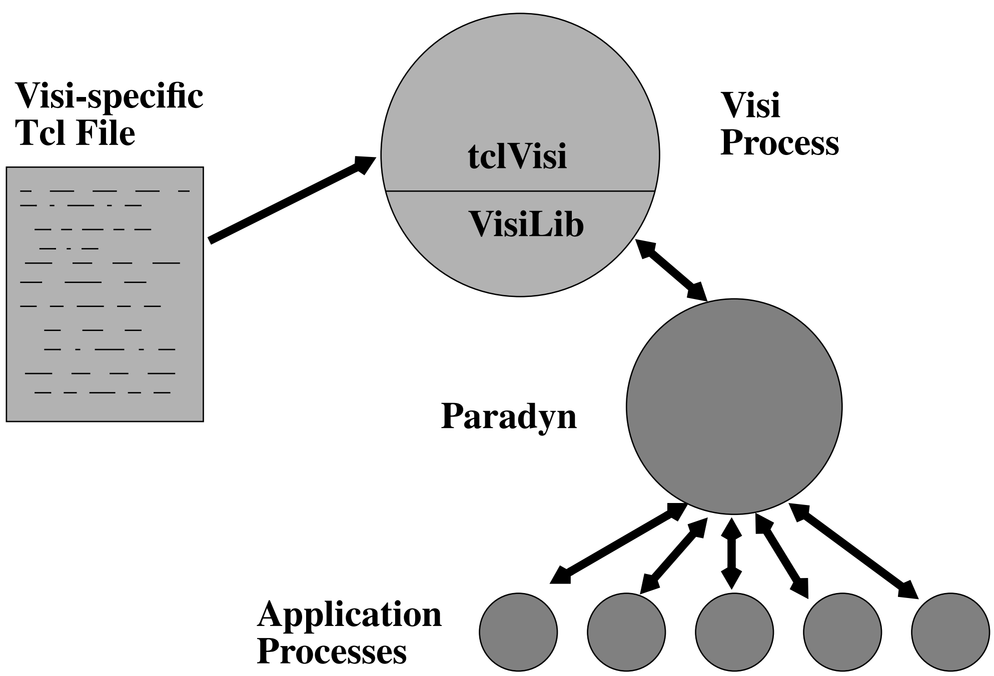

Figure 3: TclVisi

2.3.1 Initialization

Tcl applications do not need to explicitly call the *visi_Init* VisiLib routine. This routine is called in *tclVisi's* main routine.

2.3.2 Paradyn events

To be notified of a Paradyn event, a tcl visualization writer needs to implement a `Dg` command corresponding to the event. The `visi_RegistrationCallback` VisiLib routine does not need to be called by the tcl visualization writer; this routine is called by tclVisi. Instead, the visualization writer needs to write `Dg` commands for any Paradyn event in which the visualization is interested. The behavior of the command is left to the visi writer. All `Dg` command routines take an integer argument that has meaning only for the DATAVALUES, PHASESTART, and PHASEEND events. Figure 4 lists the prototypes for the `Dg` commands; the description of the event types is given in Figure 2.

`proc DgDataCallback {lastBucket}`	callback for DATAVALUES event
`proc DgInvalidCallback {}`	callback for INVALIDMETRICSRESOURCES event
`proc DgConfigCallback {}`	callback for ADDMETRICSRESOURCES event
`proc DgPhaseStartCallback {phaseid}`	callback for PHASESTART event
`proc DgPhaseEndCallback {phaseid}`	callback for PHASEEND event
`proc DgPhaseDataCallback {}`	callback for PHASEDATA event
`proc DgFoldCallback {}`	callback for FOLD event
`proc DgParadynExitedCallback {}`	callback for PARADYNEXITED event

Figure 4: Dg commands for Paradyn event callbacks

2.3.3 Getting data from the data grid

TclVisi provides a tcl command called `Dg` that a visualization process may call to get performance information. To use tclVisi, a programmer writes a tcl application and makes calls to DataGrid commands (`Dg` commands). Figure 5 contains the set of commands provided by the tcl interface. Metric identifiers (mids), resource identifiers (rids), and phase identifiers (pids) are integers between 0 and the number of metrics, or the number of resources, or the number of phases, in the DataGrid.

2.3.4 Calls to Paradyn

The tclVisi interface provides `Dg` commands for calling Paradyn. Figure 6 lists these commands and provides a brief description of their function.

`Dg nummetrics`	returns the number of currently defined metrics
`Dg metricname <mid>`	returns the name of metric <mid>
`Dg metriclabel <mid>`	returns the units label of the metric <mid>
`Dg metricavelabel <mid>`	returns the AVE units label of the metric <mid>
`Dg metricsumlabel <mid>`	returns the SUM units label of the metric <mid>
`Dg numresources`	returns the number of currently defined resources
`Dg resourcename <rid>`	returns the name of resource <rid>
`Dg valid <mid> <rid>`	returns 1 if a time histogram exists for metric <mid> and resource <rid>, otherwise returns 0
`Dg enabled <mid> <rid>`	returns 1 if data collection has been enabled for this <mid> and <rid> combination
`Dg value <mid> <rid> <bin>`	returns a time histogram data value associated with metric <mid>, resource <rid>, bucket <bin>
`Dg sum <mid> <rid>`	returns the sum of all buckets of the time histogram of metric <mid> resource <rid>
`Dg average <mid> <rid>`	returns the average of all time histogram buckets for metric <mid> resource <rid>
`Dg numbins`	returns the number of buckets in time histograms
`Dg binwidth`	returns the time histogram bucket width
`Dg lastbucket <mid> <rid>`	returns the last time histogram bucket containing performance data
`Dg firstbucket <mid> <rid>`	returns the first histogram bucket with valid data
`Dg myphasename`	returns the name of the visi's phase
`Dg myphasestartT`	returns the start time of the visi's phase
`Dg myphasehandle`	returns the unique identifier for the visi's phase
`Dg numphases`	returns the number of phases currently defined
`Dg phasename <pid>`	returns the name of phase <pid>
`Dg phasestartT <pid>`	returns the start time of phase <pid>
`Dg phaseendT <pid>`	returns the end time of phase <pid>

Figure 5: tclVisi Dg service commands

2.4 Using trace data with VisiLib

`Dg start`	start new histograms (a request to Paradyn to display menus to select new metric and resource combinations)
`Dg stop <mid> <rid>`	stop data collection for a histogram
`Dg phase <phasename> <with_pc> <with_visis>`	start a new phase

Figure 6: tclVisi commands that call the Paradyn process

Paradyn's support for trace data streams allows Paradyn to move trace data from applications to visis. Visis are used as analysis and/or display tools for the trace data. The support is based on a dynamic tracing model where Paradyn does not interpret the semantics of the trace data; Paradyn supports moving raw trace data from applications to visis.

To use trace data, some code needs to be added to the application process (the trace producer), and to the visi (the trace consumer). For the trace producer side, a trace metric definition must be added to a Paradyn configuration file. A metric definition is written using Paradyn's metric description language (MDL). MDL is described in more detail in the ***Paradyn User's Guide***. The trace metric should be defined so that it calls code in the application that will call DYNINSTgenerateTraceRecord with parameters specifying the metric ID number and the trace data flag (TR_DATA). For example, Figure 7 shows the metric definition for a metric that computes the total number of pvm_send calls. The `traceSend` function (shown in Figure 8) needs to be compiled and linked into the application. Note that `$globalId`, which is the metric identifier, is transferred along with a trace data record all the way to a trace visi by setting it to the first parameter of the `DYNINSTgenerateTraceRecord`. If there is more than one record type for a metric, then each type must be included in the record. The trace data consumer accesses trace data by calling a VisiLib function described in Section 2.2.3.7. Section 5 contains an example application using the trace data interface in VisiLib.

```
base is void { // neither a counter nor a timer is needed
      foreach func in pvm_msg_send {
            append preInsn func.entry (* traceSend($globalId, 1); *)
      }
}
```

Figure 7: Example Trace Data Metric Definition

3 COMPILING AND LINKING A VISI APPLICATION

In order to compile and link a visi, the VisiLib header file `visualization.h` must be included in the visi's source code. Also, the visi must be linked with VisiLib and with the Paradyn Utility library. For visis that are compiled using gcc, the visi must additionally be linked with the C++ standard library (stdc++). Figure 9 shows a portion of a Makefile for a visi (`xtext`).:

```
void traceSend (int globalId, int type) {
      struct _traceRecord  record;
      record.type = type;
      /* assuming that  _traceRecord  has the type field. */
      DYNINSTgenerateTraceRecord(globalId, TR_DATA,
                                sizeof(struct_traceRecord),
                                &record, 0, 0.0, 0.0);
}
```

Figure 8: traceSend routine.
This is added to the trace producer (the application's code).

```
TO_VISI = ../../../../
CC = g++

IFLAGS  = -I. -I$(TO_VISI)visi/h
LIBDIR = -L$(TO_VISI)../lib/$(PLATFORM)
CFLAGS = -O -g $(IFLAGS)

# VisiLib and Paradyn Utility Library (add -lstdc++ if using gcc)
LIBS = $(LIBDIR) -lvisi -lpdutil
LIBS += -lXaw -lXext -lXmu -lXt -lX11 -lm

SRCS = ../src/xtext.C
OBJS = xtext.o

xtext: $(OBJS)
      $(CC) -o xtext $(LFLAGS) $(OBJS) $(LIBS)
$(OBJS): $(SRCS)
      $(CC) $(CFLAGS) -c $(SRCS)
```

Figure 9: Example Makefile for a Visi

4 ADDING A VISI DESCRIPTION TO A PCL FILE

Once a visualization has been modified to use the VisiLib interface, an entry for it needs to be added to a Paradyn Configuration Language (PCL) file. PCL files are read by the Paradyn process as part of paradyn start-up. Paradyn uses the information in these files to create menu entries for visualizations and to start visualization processes that have been selected by the Paradyn user. Figure 10 shows the syntax for a PCL file visualization entry, followed by two examples.

All visualization entries start with the key word `visi`, followed by a name for the visualization that will be used as a visualization menu entry. The body of the definition (between "{" and "}") must contain the key word `command` followed by the command string for starting the visualization. The command string contains the command and the arguments to start the visualization process. The body of the definition may optionally specify a `force` value of 1 and/or a positive `limit` value. Visualizations with a force value of 1 will be created without menuing for metrics and resources first. The default behavior is that Paradyn initiates menuing before starting the visual-

Syntax for PCL File *Visi* Entry:

```
visi <name> {
      command <command string>;
      [force <int>;]
      [limit <int>;]
}
```

PCL File Example *Visi* Entries:

```
visi MyBarChart {
      command "barChart";
}
visi MyPhaseTable {
      command "tclVisi -f phasetbl.tcl";
      force 1;
}
```

Figure 10: PCL Visi entry syntax and examples

ization process. An example of a visualization that would specify a force value of 1 is the phase table visualization (this visualization is discussed in Section 5.2). The phase table is used to display phase information for all the phases in the system. Since this visualization is not used to display performance data for metric/resource pairs, it does not make sense to do menuing for metrics and resources before the phase table process is started. The `limit` option specifies an upper bound on the number of metric/resource pairs that the visi can have enabled at one time. If this field is not specified, or if it has a non-positive value, then there is no upper bound.

In the first example in Figure 10, a barChart visualization has been defined, such that its menu label will be **MyBarChart** and such that the paradyn process will initiate metric and resource menuing before starting the barChart process. The second example defines a phase table visualization that uses the tclVisi interface. In this example, the `force` option set to 1 so menuing will not occur before the visualization process is started.

5 EXAMPLES

This section contains two simple examples of visualization applications that use the VisiLib interface. The first uses the VisiLib interface directly, and the second uses the tclVisi interface.

5.1 Example that uses the VisiLib interface

The example application described below is a simple X application taken from the XConsortium's suite of example programs[2]. The application uses the X Toolkit and the Athena Widget set. X

2. "Athena Widget Set - C Language Interface, X Window System, X Version 11, Release 5", Chris Peterson, MIT X Consortium.

Toolkit commands are prefixed by *Xt* and the header files for the Athena Widget set are in *"X11/Xaw"*.We modified the application to use the VisiLib interface; Figure 11 shows this modified application. It consists of a set of command widgets, which when selected, invoke some action on the text widget that is used to display DataGrid bucket values. The original program consisted of just the *Clear, Print,* and *Quit* command widgets that would invoke actions on the text widget.

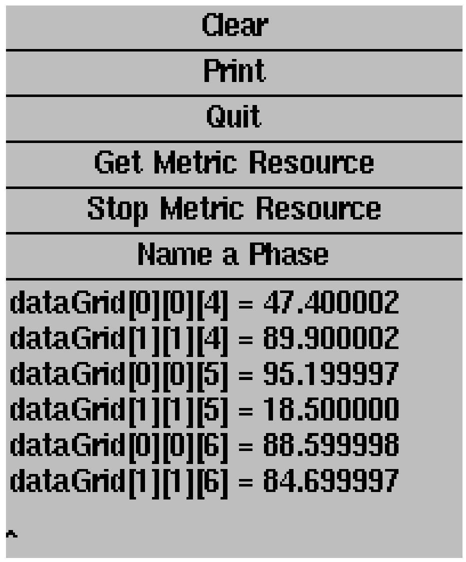

Figure 11: Example Visualization

5.1.1 Steps to modify the example application

In order to use VisiLib, a visualization application must contain code that initializes VisiLib, that makes calls to Paradyn, and that handles Paradyn events. We describe this code below as a set of steps. By following these steps, a visualization writer can create a visualization that accesses Paradyn performance data. In Section 5.1.2 we provide code fragments from the example application that illustrate these steps.

STEP 1: Add a call to `visi_Init` in the application's main routine.

STEP 2: Write callback routines for Paradyn events, and register them by calling `visi_RegistrationCallback` in the application's main routine.

STEP 3a: Create a mechanism to make calls to Paradyn. Typically this is done by adding some type of menuing widget that will make the call to Paradyn when selected.

STEP 3b: Register callback routines on the menuing widgets created in step 3a.

STEP 4: Add the PDSOCKET returned by `visi_Init` as a source of input for the application and add the VisiLib routine `visi_callback` as a callback routine for the file descriptor.

STEP 5: The final step is to enter the main loop of the application.

5.1.2 Source code from the example application

The code fragment shown in Figure 12 consists of the main loop of this application with some of the error handling code removed. Also, some of the uninteresting parameters to Xt functions have been removed to make the code more readable.

To use VisiLib routines, the VisiLib header file *visualization.h* must be included. The other header files included in this code are Xlib, X-Toolkit, and Athena Widget set header files. Since this application uses the main loop provided by X-Toolkit, all widget creation and other initialization must be done in the `main` routine before the application calls `XtAppMainLoop`. The following is a description of how we applied the steps listed in Section 5.1.1 to this application:

STEP 1: The first section of code in the main routine initializes the visualization library by calling the `visi_Init` routine. This routine returns a PDSOCKET through which the visualization can communicate with the paradyn process.

STEP 2: Callback routines were written for the following Paradyn events: FOLD, ADDMETRICSRESOURCES, DATAVALUES, and PHASEDATA. These routines are registered by making calls to `visi_RegistrationCallback` with arguments indicating the event and the handler. When, for example, data values arrive from the paradyn process, the `dataHandler` routine will be called by a routine in the visualization library. (The `dataHandler` routine is shown in Figure 13)

STEP 3a: A mechanism for the visualization to make calls to Paradyn is added. In this application we have added three command widgets (getMr, stopMr, phaseN) to do this. By adding calls to `XtCreateManagedWidget` we created command widgets to get new metrics and resources, to stop data collection for a metric/resource pair, and to start a new phase.

STEP 3b: We registered callback routines on the command widgets created in step 3a, by adding calls to the XToolkit function `XtAddCallback`. Calls to this routine take a command widget argument and an action routine that will be called when the command widget is selected. Figure 14 lists the command action code associated with these widgets.

STEP 4: The visualization process must register this file descriptor as a source of input. In this application the file descriptor is added as a source of input by calling the `XtAppAddInput` routine before entering the main loop. The VisiLib routine `visi_callback`, a parameter to `XtAppAddInput`, is the callback routine for input on the file descriptor. The routine to register the file descriptor will vary from toolkit to toolkit.

STEP 5: Before entering the application's main loop, we make a call to `XtRealizeWidget` which is an XToolkit routine that binds action names to procedures, sets widget's attributes, and maps the application main window. We then enter the application's main loop by calling `XtAppMainLoop`.

```
// include files from original application
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/AsciiText.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Cardinals.h>

// VisiLib include file
#include “visi/h/visualization.h”

main(int argc, char **argv) {
      // code to initialize the application
      Widget toplevel = XtAppInitialize (&app_con, ...);


      // Step 1: call visi_Init
      PDSOCKET sock = visi_Init();
      if (sock == PDSOCKET_ERROR)
            exit(-1);


      // Step 2: register callbacks for Paradyn events
      //    these routines are called when the associated Paradyn
      //    event occurs
      int ok;
      ok = visi_RegistrationCallback(ADDMETRICSRESOURCES,addMRHandler);
      ok = visi_RegistrationCallback(DATAVALUES, dataHandler);
      ok = visi_RegistrationCallback(FOLD, foldEventHandler);
      ok = visi_RegistrationCallback(PHASEDATA, phaseEventHandler);


      // create the original application’s widgets
      Widget paned = XtCreateManagedWidget(“panned”, ...);
      Widget clear = XtCreateManagedWidget(“Clear”, ...);
      Widget print = XtCreateManagedWidget(“Print”, ...);
      Widget quit = XtCreateManagedWidget(“Quit”, ...);


      // Step 3a: create command widgets that will make calls to Paradyn
      Widget getMR = XtCreateManagedWidget(“Get Metric Resource”, ...);
      Widget stopMR = XtCreateManagedWidget(“Stop Metric Resource”,...);
      Widget phaseN = XtCreateManagedWidget(“Name a Phase”, ...);


      // create text widget (from original application)
      Widget text = XtCreateManagedWidget(“text,...);
```

Figure 12: Main routine for example application

```
    // add callbacks for original widgets (from original application)
    XtAddCallback(clear, XtNcallback, ClearText, (XtPointer)text);
    XtAddCallback(print, XtNcallback, PrintText, (XtPointer)text);
    XtAddCallback(quit, XtNcallback,QuitProgram, (XtPointer)app_con);

    // Step 3b: Add callbacks to widgets that make calls to Paradyn
    //    These are callback routines that are called when the
    //    associated command widget is selected
    XtAddCallback(getMR, XtNcallback, GetMetsResUpcall, ...);
    XtAddCallback(stopMR, XtNcallback, StopMestResUpcall, ...);
    XtAddCallback(phaseN, XtNcallback, StartPhaseUpcall, ...);

    // Step 4: register the VisiLib routine “visi_callback” on events
    // associated with the file descriptor returned by visi_Init
    XtAppAddInput(app_con, fd, XtInputReadMask, visi_callback, ...);

    // Step 5: enter main loop
    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_con);
}
```

Figure 12: Main routine for example application

```
int dataHandler (int bucketNum){
      // display value of bucketNum for all valid DataGrid elements
      int numMets = visi_NumMetrics();
      int numRes = visi_NumResources();
      for (int i = 0; i < numMets; i++){
            for (int j = 0; j < numRes; j++){
                  if (visi_Valid(i,j))
                        DrawData(visi_DataValue(i,j,bucketNum));
            }
      }
}
```

Figure 13: Callback routine for DATAVALUES event

```
static void GetMetsResUpcall(Widget w, XtAppContext ac, XtPointer p) {
      visi_GetMetsRes(0,0); // call VisiLib routine
}
static void StopMetsResUpcall(Widget w, XtAppContext ac, XtPointer p) {
      visi_StopMetRes(0,0); // call VisiLib routine
}
static void StartPhaseUpcall(Widget w, XtAppContext ac, XtPointer p) {
      visi_DefinePhase(0,0,0); // call VisiLib routine
}
```

Figure 14: Code to make calls to Paradyn

5.2 Example application using the tcl interface to VisiLib

The example shown in Figure 15 is a visualization application that uses the tcl interface to VisiLib. It is used to display Paradyn's phase information. For applications of this type, the programmer writes a tcl application that makes calls to DataGrid (`Dg`) commands. The tcl interface handles calling the VisiLib initialization routine and registering callback routines on Paradyn events. The tcl application writer needs only to implement the `Dg` callback routines for which the application is interested (the prototypes for these routines are listed in Figure 4), and to add calls to `Dg` commands that make calls to Paradyn (these `Dg` commands are listed in Figure 6).

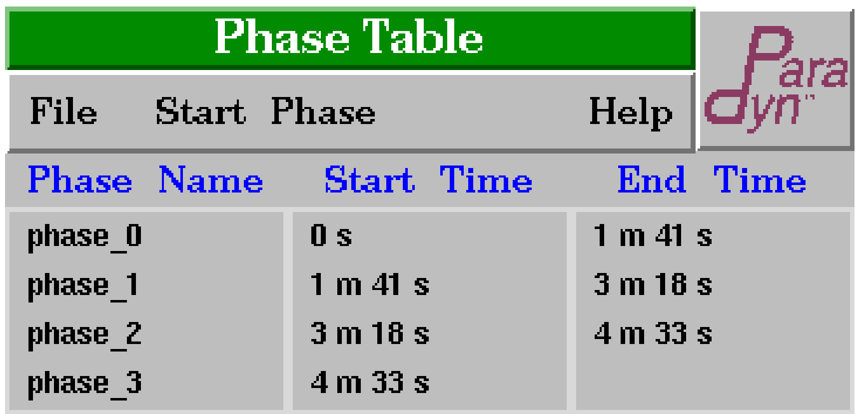

Figure 15: Example visualization that uses the Tcl interface to VisiLib

Because most of the work required to write a visualization that uses VisiLib is done in tclVisi, tcl visualization writers need only implement the two steps listed below:

STEP 1: Implement `Dg` callback commands for any Paradyn event of which the visualization wishes to be notified. The complete list of `Dg` commands for Paradyn events are listed in Figure 4

STEP 2: Add calls to `Dg` commands where needed. Typically this is done by adding some type of menuing widget that will make the call to Paradyn when selected. In the example application, this is done by creating a menu button labeled **StartPhase**, with the `-command` option "`Dg phase`".

5.2.1 Source code from the example application

TclVisi takes care of registering callbacks on Paradyn events by calling the VisiLib `visi_RegistrationCallback` routine. Tcl visualization writers need only implement the `Dg` callback commands. Since the phase table visualization is used solely to display Paradyn phase information, we only wrote `Dg` callback commands for the STARTPHASE, ENDPHASE, and PHASEDATA Paradyn events. For the phase table we implemented `DgPhaseStartCallback`, `DgPhaseEndCallback`, and `DgPhaseDataCallback`; the tcl code for our implementation of these is shown in Figure 16. These `Dg` callback implementations make calls to `StartUpdate`, `NameUpdate`, and `EndUpdate`. The tcl code for `StartUpdate` is show in Figure 17. This code handles displaying the start time of the phase associated with `phaseId`. The call "`Dg phasestartT`

`$phaseId"`, is a DataGrid command to get the start time for the phase with the phase identifier `phaseId`.

```
#
# tclVisi registers this routine as callback for PHASESTART event
#
proc DgPhaseStartCallback {phaseId} {
      # adds new phase's name to the "Phase Name" column
      NameUpdate $phaseId
      # adds new phase's start time to the "Phase Start" column
      StartUpdate $phaseId
      return
}
#
# tclVisi registers this as a callback for PHASEEND event
#
proc DgPhaseEndCallback {phaseId} {
      # adds current phase's end time to the "Phase End" column
      EndUpdate $phaseId
      return
}
#
# tclVisi registers this as a callback for PHASEDATA event
#
proc DgPhaseDataCallback {} {
      # make DataGrid call to get the max number of phases
      set max [expr int ([Dg numphases])]
      for {set phasecount 0} {$phasecount < $max} {incr phasecount} {
            NameUpdate $phasecount
            StartUpdate $phasecount
            if {$phasecount < [expr $max - 1]} {
                  EndUpdate $phasecount
            }
      }
      return
}
```

Figure 16: Dg callback routines for PhaseTable visualization

The segment of code in Figure 18 consists of most of the tcl code to create the Phase Table visualization. This code fragment contains tcl code for implementing step 2. Its code contains one tcl command that is used to make a call to Paradyn; this call is to start a new phase. This command creates a **Start Phase** menu button with a command option "`Dg phase"`. When selected, this menu button will call the `Dg phase` command.

5.3 A complete example application

In this section we provide the complete source code for an example application. The example visualization is shown in Figure 19. It is a simpler version of the application described in Section 5.1. We provide a complete listing of the source code for this application in Figure 20.

```
proc StartUpdate {phaseId} {
       global W
       global xOffset
       global yOffset
       global widthChange

       set theCanvas $W.middle.dataCanvas
       set yOffset [expr 5 + $phaseId * $widthChange]
       set theText [expr int([Dg phasestartT $phaseId])]
       $theCanvas create text $xOffset $yOffset -anchor nw \
             -fill black -justify center \
             -tag dataTag -text $theText
       return
}
```

Figure 17: Tcl code for StartUpdate

```
#
#  Create the overall frame
#
set W .table
frame $W -class Visi -width 4i -height 2i


#
#  Create the title bar, menu bar, and logo at the top
#
frame $W.top
pack $W.top -side top -fill x
frame $W.top.left
pack $W.top.left -side left -fill both -expand 1

label $W.top.left.title -relief raised -text "Phase Table" \
      -foreground white -background Green4
      pack $W.top.left.title -side top -fill both -expand true


#
#  Create the menubar as a frame with many menu buttons
#
frame $W.top.left.menubar -class MyMenu -borderwidth 2 -relief raised
pack $W.top.left.menubar -side top -fill x


#
# File menu
#
menubutton $W.top.left.menubar.file -text "File" /
      -menu $W.top.left.menubar.file.m
menu $W.top.left.menubar.file.m
$W.top.left.menubar.file.m add command -label "Close" -command Shutdown
```

Figure 18: Tcl code for PhaseTable visualization

```
#
# Step 2: Create the "Start Phase" menu button that will call
#           "Dg StartPhase" when selected
#
button $W.top.left.menubar.acts -text "Start Phase" \
      -relief flat -borderwidth 0 -highlightbackground white \
      -command "Dg phase 0 0 0"


#
#  Help menu
#
menubutton $W.top.left.menubar.help -text "Help" \
      -menu $W.top.left.menubar.help.m
menu $W.top.left.menubar.help.m
$W.top.left.menubar.help.m add command -label "Context" \
      -command HelpContext


#
#  Build the menu bar and add to display
#
pack $W.top.left.menubar.file $W.top.left.menubar.acts -side left
pack $W.top.left.menubar.help -side right


#
#  Build the logo
#
label $W.top.logo -relief raised -bitmap @logo.xbm -foreground HotPink4
pack $W.top.logo -side right


#
# Left portion of middle: phase name, name canvas
#
frame $W.left
pack   $W.left -side left -fill both -expand true
label $W.left.phaseName -text "Phase Name" -foreground Blue
pack  $W.left.phaseName -side top -expand false
canvas $W.left.dataCanvas -relief groove -width 1.3i
pack   $W.left.dataCanvas -side left -fill both -expand true
```

Figure 18: Tcl code for PhaseTable visualization

```
#
# Middle portion of middle: phase start time, data canvas
#
frame $W.middle
packpack   $W.middle –side left –fill both –expand true
label $W.middle.phaseStart –text “Start Time” –foreground Blue
pack  $W.middle.phaseStart –side top –expand false
canvas $W.middle.dataCanvas –relief groove –width 1.3i
pack   $W.middle.dataCanvas –side left –fill both –expand true


#
# Right portion of middle: phase end time, data canvas
#
frame $W.right
pack   $W.right –side left –fill both –expand true
label $W.right.phaseEnd –text “End Time” –foreground Blue
pack  $W.right.phaseEnd –side top –expand false
canvas $W.right.dataCanvas –relief groove –width 1.3i
pack   $W.right.dataCanvas –side left –fill both –expand true


#
# display everything
#
pack append . $W {fill expand frame center}
wm minsize . 250 100
wm title . “Phase Table”
```

Figure 18: Tcl code for PhaseTable visualization

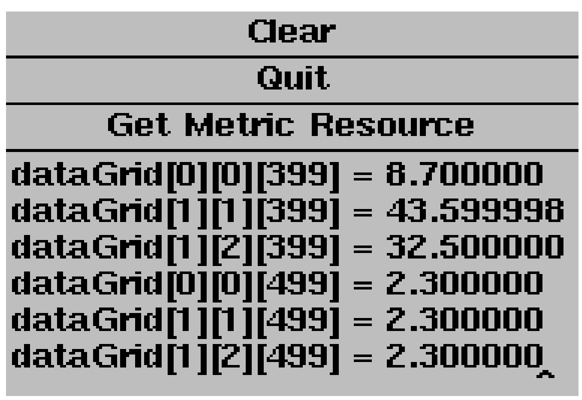

Figure 19: Complete example application

```
/*
* $XConsortium: xtext.c,v 1.16 91/05/16 14:56:23 swick Exp $
*
* Copyright 1989 Massachusetts Institute of Technology
* Permission to use, copy, modify, distribute, and sell this software and its
* documentation for any purpose is hereby granted without fee, provided that
* the above copyright notice appear in all copies and that both that
* copyright notice and this permission notice appear in supporting
* documentation, and that the name of M.I.T. not be used in advertising or
* publicity pertaining to distribution of the software without specific,
* written prior permission. M.I.T. makes no representations about the
* suitability of this software for any purpose. It is provided "as is"
* without express or implied warranty.
*
* M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL M.I.T.
* BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
* WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION
* OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
* CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

   // include files
   #include <stdio.h>
   #include <X11/Intrinsic.h>
   #include <X11/StringDefs.h>
   #include <X11/Xaw/AsciiText.h>
   #include <X11/Xaw/Command.h>
   #include <X11/Xaw/Paned.h>
   #include <X11/Xaw/Cardinals.h>
   #include "visi/h/visualization.h"

   // global variables
   XtAppContext app_con;
   Widget toplevel, paned, clear, quit, text, getMR;


   String fallback_resources[] = {
         "*background: Grey",
         "*foreground: Black",
         "*font: *-Helvetica-*-r-*-12-*",
         "*input:                              True",
         "*showGrip:                           off",
         "?.?.text.preferredPaneSize:          200",
         "?.?.text.width:                      200",
         "?.?.text.textSource.editType:        edit",
         "?.?.text.scrollVertical:             whenNeeded",
         "?.?.text.scrollHorizontal:           whenNeeded",
         "?.?.text.autoFill:                   on",
         "*clear*label:                        Clear",
         "*quit*label:                         Quit",
         NULL
   };
```

Figure 20: Complete source code for example application

```
// callback routine for DATAVALUES
int dataValuesCallback(int dummy){
      Arg args[1];
      XawTextPosition pos;
      XawTextBlock tb;
      char buf[100];

      XtSetArg(args[0], XtNinsertPosition, &pos);
      XtGetValues(text, args, ONE);

      int noMetrics = visi_NumMetrics();
      int noResources = visi_NumResources();

      for(int i=0; i < noMetrics; i++){
         for(int j=0; j < noResources; j++){
            if(visi_Valid(i,j)){
                  if (!(isnan(visi_DataValue(i,j,dummy)))){
                        sprintf(&buf[0],"%s%d%s%d%s%d%s%f\n",
                           "dataGrid[",i,"][",j,"][",dummy,"] = ",
                           visi_DataValue(i,j,dummy));
                  } else {
                        sprintf(&buf[0],"%s%d%s%d%s%d%s\n",
                           "dataGrid[",i,"][",j,"][",dummy,"] = NaN");
                  }

                  int size = strlen(buf);
                  tb.firstPos = 0; tb.length = size;
                  tb.ptr = buf;    tb.format = FMT*BIT;
                  XawTextReplace(text,pos,pos+size-1, &tb);
                  pos += size;
                  XtSetXtSetValues(text,args,ONE);
            }
      } }
      return 1;
}

// callback routine for PARADYNEXITED event
int exitedCallback(int dummy){ }

// makes call to Paradyn
static void GetMetsResCallback(Widget w, XtAppContext app_con,
                              XtPointer call_data)
{
      visi_GetMetsRes(0,0);
}

// clears text out of the text widget
static void ClearText(Widget w, XtPointer text_ptr, XtPointer call_data)
{
      Widget text = (Widget) text_ptr;
      Arg args[1];
      XtSetArg(args[0], XtNstring, "");
      XtSetValues(text, args, ONE);
}
```

Figure 20: Complete source code for example application

```
// exits the program
static void QuitProgram(Widget w, XtAppContext app_con,
                        XtPointer call_data)
{
      XtDestroyApplicationContext(app_con);
      exit(0);
}


main(int argc, char **argv) {
      Arg args[1];

      // code to initialize the application
      Widget toplevel = XtAppInitialize (&app_con, "Xtext",NULL,ZERO,
                        &argc, argv, fallback_resources, NULL, ZERO);

      if (argc != 1) exit(1);

      // call visi_Init
      int fd = visi_Init();
      if (fd < 0)
            exit(-1);

      // register callbacks for Paradyn events
      int ok = visi_RegistrationCallback(DATAVALUES,
                                    dataValuesCallback);
      ok = visi_RegistrationCallback(PARADYNEXITED, exitedCallback);

      // create the original application's widgets
      Widget paned = XtCreateManagedWidget("panned",panedWidgetClass,
                                    toplevel, NULL, ZERO);
      Widget clear = XtCreateManagedWidget("Clear",commandWidgetClass
                                    paned, NULL, ZERO);
      Widget quit = XtCreateManagedWidget("Quit",commandWidgetClass,
                                    paned, NULL, ZERO);

      // command widget that will make calls to Paradyn
      Widget getMR = XtCreateManagedWidget("Get Metric Resource",
                        commandWidgetClass, paned, NULL,ZERO);

      // create text widget
      Widget text = XtCreateManagedWidget("text,asciiTextWidgetClass,
                                    paned, NULL, ZERO);

      // add callbacks for original widgets
      XtAddCallback(clear, XtNcallback, ClearText, (XtPointer)text);
      XtAddCallback(quit,XtNcallback,
                  QuitProgram,(XtPointer)app_con);

      // Add callbacks to widgets that make calls to Paradyn
      XtAddCallback(getMR, XtNcallback, GetMetsResUpcall,
                  (XtPointer) app_con);
```

Figure 20: Complete source code for example application

```
        // register the VisiLib routine "visi_callback" on fd
        XtAppAddInput(app_con, fd, XtInputReadMask,
                     (XtInputCallbackProc)visi_callback, text);

        // enter main loop
        XtRealizeWidget(toplevel);
        XtAppMainLoop(app_con);
}
```

Figure 20: Complete source code for example application

5.4 An example application using the trace data interface in VisiLib

For a visi to receive trace data, it must register a callback routine on the TRACEDATAVALUES event.; a call to `visi_RegistrationCallback` with the TRACEDATAVALUES flag and the callback routine as parameters must be added to the visi code. For example:

```
visi_RegistrationCallback(TRACEDATAVALUES, Dg2NewTraceDataCallback);
```

In this example, `Dg2NewTraceDataCallback` is called for every trace record arriving at the visi. The function needs to be written appropriately to compute a trace metric on the visi. Figure 21 shows a sample callback function which counts the number of pvm_send and pvm_recv calls. Trace data can be sent from application processes to a trace visi by enabling data collection for a trace metric. A trace metric must be added to a Paradyn configuration file prior to running Paradyn. This is described in Section 2.4.

5.5 Other examples

The VisiLib sources contain a test directory with sample visualizations that have been modified to use VisiLib. Each of the application's source files contain comments on how the original application was modified to use VisiLib. Also, in this directory is a psuedo-paradyn process that can be used to test visualization applications that use either VisiLib or tclVisi. These directories contain README files that explain how to run these applications.

6 COMMENTS AND QUESTIONS

The source distribution of VisLib and tclVisi contain README files describing how to use the interface; the VisiLib sources contain a test suite with example visualization applications that have been modified to use the visualization interface. All questions and comments regarding VisiLib and tclVisi should be directed to `paradyn@cs.wisc.edu`.

■

```
#include "visi/src/visiTypesP.h"
struct _typeExtract {
      int   type; // 1 for pvm_send and 2 for pvm_recv
};
typedef struct _typeExtract typeExtract;
static int pvmSendNum = 0;
static int pvmRecvNum = 0;
int Dg2NewTraceDataCallback(int) {
      int metricIndex = visi_TraceDataValues()->metricIndex;
      char *namePtr =  visi_MetricName(metricIndex);
      cout << "The metric name :" << namePtr << endl;
      char *sp = visi_TraceDataValues()->dataRecord->getArray();
      typeExtract *b = (typeExtract *)sp;
      if (b->type == 1) {
            pvmSendNum++;
            cout << "@@@ " << pvmSendNum << "th pvm_send call" << endl;
      } else if (b->type == 2) {
            pvmRecvNum++;
            cout << "@@@ " << pvmRecvNum << "th pvm_recv call" << endl;
      } else {
            cout << "@@@ type unknown" << endl;
      }
      return TCL_OK;
}
```

Figure 21: Example sample callback function for trace data.
This counts the number of pvm_send and pvm_recv calls.