GDBTk - Integrating Tcl/Tk into a recalcitrant command-line application.

Jim Ingham
Cygnus Solutions
jingham@cygnus.com

Abstract:

This paper will describe the some of the lessons we learned embedding Tcl/Tk into the Gnu Debugger (GDB). GDB is a command-line application, and there were a number of problems which this caused, particularly when using Tk within the application. We will describe a few of these problems, and the solutions we came up with. The paper will also give a brief overview of GDBTk itself.

Introduction

GDBTk started as a skunkworks project by Stu Grossman in 1994. There had been a number of other GUI interfaces to gdb before this (xgdb, xxgdb, the Emacs interface, and later DDD). GDBTk was distinguished from them by having the GUI actually linked with the debugger. Most of the other GUI's for GDB spawned GDB as a child process and then used the GDB command language to drive GDB, and parsed the output to get information back out of it. This method overcomes several significant problems with living in the same application as GDB, but has a number of drawbacks.

First among the drawbacks, there are some operations which are too slow when the information has to be printed by GDB, and then parsed in a separate program. Examples of this are presenting a variable display when there are many complicated structures in scope, or performing disassembly (or mixed source/assembly display) on large functions. Second, GDB has a lot of knowledge about the executable which it does not necessarily make available through the command line, or does not make available in a compact form. Efficient listing of backtraces, or function lookup are several examples of this. Thirdly, and perhaps most importantly, having a real scripting language embedded into GDB has persuasive advantages all on its own that fully justify the effort.

In the first section of this paper, I will describe the general structure of GDBTk. In the second, I will lay out a few of the problems we had in working with GDB, and the solutions we came up with, hopefully drawing some lessons for others faced with a similar task. Then in the last section, I will give a brief tour of the current state of GDBTk, and point out some of the places where the implementation in Tcl has caused particular problems, as well as some instances where the use of Tcl really shines

Since the initial version by Stu Grossman, there have been two other GDB GUI's produced at Cygnus. Before Tk was available on Windows, Steve Chamberlin wrote the (ill-fated) WinGDB, an MFC based Windows application. How-

ever, as soon as a cross-platform version of Tcl/Tk was available, development on this was stopped, and Martin Hunt, Keith Seitz, Tom Tromey and Ian Lance Taylor developed a much enhanced version of the original GDBTk code. I joined Cygnus to work on this project after the Sun Tcl effort folded in March 1998. Since that time GDBTk has seen considerable improvement, and is also included as the debugger interface in the CodeFusion IDE[1] from Cygnus. In August of 1999, GDBTk (now called Insight) was released under GPL to the net[2].

GDBTk Internals:

GDBTk is constructed in four major parts. The first is GDB itself. Most of the core functionality of GDBTk resides in GDB. We try not to duplicate code that is handled in the GDB core wherever possible, though, as I will mention below, there are cases where this is unavoidable. Also, although, due to the strictures of the FSF which owns the actual copyrights to GDB, we cannot actually introduce Tcl code into the GDB core, GDB itself contains a number of hooks - callouts to client-defined functions - that GDBTk uses to find out about interesting events in the debugger.

The second part of GDBTk is its implementation of the hook functions. These hooks fall into two broad categories. One type provides the GUI with notification of events that happen outside of its control in GDB. An example of this is that, when GDB hits a temporary breakpoint, it deletes the breakpoint. The GUI needs to be notified of this fact, so that it can remove the breakpoint from the breakpoint window. The other main function of the hooks is to allow GDBTk to keep in synch with the standard GDB command language console. Anything significant that can be changed in the console is given a hook function, and that is how the GUI is notified of the changes the user makes there.

The third part of GDBTk's implementation is a Tcl extension that provides access to many of GDB's internal API's. On the simplest level, we have the `gdb_cmd` command, which just passes its argument to the GDB command parser, and returns whatever the GDB command would have written to stdout. At the same time, we introduce a command: `tk` into the GDB command interpreter, so we can run Tcl code from the GDB console window.

When you want to instruct GDB to perform some action - set or delete a breakpoint, open an executable file, detach from a running program, gdb_cmd is all you need. The output is either very simple or non-existent, so duplicating the functionality in Tcl would serve no purpose.

However, for commands that get information out of GDB - getting variable values, or reading symbols out of the symbol table, there is either no GDB equivalent, or the equivalent has output which is in an inconvenient form. In these cases, we have added Tcl commands that perform the tasks. Some examples of given below:

```
gdb_listfiles..............Lists the files in the current
   executable.

gdb_listfuncs file.........List the functions in "file"

gdb_loc symbol.............Returns a complete specifica-
   tion (file, line number and pc) for the symbol "sym-
   bol".

gdb_get_regs...............Returns a list containing the
   current register values.
```

There are 49 commands in the gdbtk command set.

---

1. See http://www.cygnus.com/codefusion
2. See http://.sourceware.cygnus.com/insight

The most interesting set of C-based Tcl commands in GDBTk are the ones that deal with listing the variables in the current scope. This was an area where parsing the GDB output was difficult and slow. Keith Seitz created a set of Tcl commands that return the blocks in a function, the variables in each block and then a Tk-like set of commands that create Tcl access commands for each variable, or indeed for any valid expression. The access commands allow you to access the variable in the executable - query and change its value, get its type, dereference pointers, and get all the children of a structure or all the instance data of a C++ object. Using these, we are able to construct a local variables window which can update itself with no visible lag on any reasonable system, even in test cases with 600 variables in the current scope.

The final component of GDBTk is the actual GUI implementation code. We chose IncrTcl for the implementation, since this we knew this was going to be a big project, and we felt this was the best way to manage the complexity. Since much of the code was written originally for Itcl1.5, we were much less ambitious about the class hierarchies than we could be if we were writing it with Itcl3.0. We have a general ManagedWin base class that all the windows derive from. This allows us to coordinate the windows, and is a convenient factory for the windows that we want to be unique. We also have generic dialog classes, etc. And each window type is a separate class. Needless to say, we are very happy with IncrTcl, it has made the project much more maintainable.

We also use a number of other components from the Net: a somewhat hacked version of Brian Oakely's ComboBox[3], a number of the IncrWidgets[4], and TkTable[5]. We also use Tix, though given that it is currently maintained, and is showing its age, we have been backing out all the uses of it that we can. The only GUI element that we can currently find only in Tix is a Tree-driven table widget. The TkTable does well for things like the memory display. The BLT[6] her widget makes a good tree. But for the variable window, we need both the hierarchical Tree structure to represent structures, and a table to list variable types and values in columnar form. The value elements also need to be editable. We tried the experiment of coupling BLT's hier widget with the TkTable, but had no success in getting the two to expand in synch. The visual artifacts were very distracting.

What about GDB makes this project difficult?

GDB was designed to be a command line application, and was never intended either as a callable library, or as a scriptable application. The difficulties this causes fall into two main categories: lack of a real "callable interface", and the blocking behavior typical of a command line application. I will next describe these features in turn:

1.) Lack of a "Callable Interface": While GDB does have a command language, for the most part that command language lacks the notion of "return values" for command operations. Moreover, since it does not itself ever use things like the values of structures internally, it has no facility to return this information at the C level. Instead, for instance, when you want the value of a variable or expression, you call a routine that evaluates the expression, and in the course of the evaluation, prints out the type and value of the variable. If the variable is a complicated object, like a structure, the subelements and their types are recursively printed as well. The same is true for other useful bits of information like accessing the registers, the stack, and the symbols that the debugger knows about. Needless to say, this makes it very hard to write a reasonable set of Tcl commands that access the information you need to get out of GDB.

2.) Blocking Behavior: GDB spends a lot of its time waiting for the debugger target to do something interesting, whether this be hit a breakpoint, evaluate an inferior function call, or return some requested bit of data. During this time, there is not much useful that you can do with the debugger. Because of this, the original designers did not make any effort not to block while waiting. After all, if you did need to wake up the debugger, you could

---

3. See http://www1.clearlight.com/~oakley/tcl/combobox/index.html
4. See http://www.tcltk.com/iwidgets
5. http://www.purl.org/net/hobbs/tcl/capp/tkTable/tkTable.html
6. See http://www.tcltk.com/blt

always send Control-C, which would interrupt the execution and return control to the user. This is very bad for a GUI application, however, since the UI has to deal with things like repainting while the target is running.

One other minor annoyance that is worth noting is the exception handling in GDB. GDB was designed to handle all errors by setjmp/longjmp. Essentially, the program does a setjmp in the command loop at the top of the application, and if it ever runs into trouble, it just longjmps back to the top level. There are, of course, facilities to register "clean-ups" to deallocate memory etc. This mode of exception handling is very simple to implement, since you don't have to bother with returning status, and figuring out how to back out nicely, which is why it was chosen for GDB. However, it is fatal for an system like Tcl which keeps vital information on the C-stack.

The solution to this problem in GDBTk is to use a generic "call wrapper" as the command proc for all the C based Tcl commands. The actual command is passed in the ClientData field. The call wrapper stores away the old jump buffer, does some other housekeeping, does its own setjmp, and finally calls the Tcl command. This works fairly well, but in some cases is too coarse grained to allow the Tcl command to recover properly. We keep a steadily growing set of wrapped versions of useful gdb calls, and use those when it matters.

Solutions -

Callable Interface:

This is perhaps the most serious problem with GDB. The progressive accumulation of results is endemic throughout the program, and in many cases there is no other interface to the data, so you would have to start from scratch and rewrite the functionality. Doing this is complicated by the fact that many people - including DDD and the other external debuggers - depend on the exact form of the output from GDB, so you would also have to exactly replicate the current functionality, or leave both implementations in place, which is a code-maintenance nightmare. There were simply not the resources for this level of rewrite, so a different approach had to be found.

Solution 1: *puts hooks*

The first solution - used in the original GDBTk - was to search for all occurrences of the C library stdout routines, and replace them with a gdb functions that ultimately route through a single function - fputs_unfiltered. This function would just call fputs in the non-GUI case, and would call a function in Tcl-land which would accumulate the characters into the Tcl result, in the GUI case. This was natural for gdb, because it already had code to handle paging of output, so most of the printing was already routed through gdb functions.

This method, while it can be made to work, and is in fact the current base for GDBTk, has several inherent flaws. One has to do with mixing streams of output. An obvious example is that you have to strictly separate the error and output streams, or you will mix error strings into the output you are trying to parse, often causing the parsing to fail altogether, but at least presenting erroneous data in the GUI.

In some cases, this is an easily solvable problem, of course, simply use separate streams for output & error. However, this does not help you in cases where, in the course of gathering one bit of information you have to call into a function which outputs some other information. Ring all the changes on this - only SOMETIMES calls into... - and it leads to a fragile framework.

The next flaw in this method is that it forces you to parse string data. Particularly when getting the values of structures, this parsing task can be quite complicated and error prone. We have alleviated this somewhat by introducing a flag that we pass to the accumulator hook to tell it to add each new element to the result as a Tcl list element. This works sometimes, but often gdb does not accumulate the results in a particularly coherent way. Moreover, the output is not self-descriptive, so as the data is output, information about what it is lost, and has to be reconstituted in the Tcl code, once again leading to errors.

Solution 2: *Libgdb*

To solve these problems, Cygnus started to work on the "libgdb" project. This is a modest project, we are not rewriting GDB to be a true callable interface. The goal is rather to revise its output capabilities so that data can be annotated as it is produced, and so that output streams can be separated. There were two other restrictions to the design of the libgdb project, it has to replicate exactly the current gdb output, so we don't break extant scripts and clients of the command line, and it has to be adaptable to any scripting interface, since GDB is officially an FSF project, and Tcl is persona non grata in the eyes of the FSF.

The way it works is that we introduce a function table that contains another set of calls to do the printing. This as a richer set of calls than the earlier GDB set, and allow you to qualify the output as you print it. In the core gdb code, the calls appear as ui_out_..., but this is actually a #define that calls into the function table of the currently installed output builder. So for instance, the old version of printing breakpoint information looked like (printf_filtered is one of the original gdb printing functions):

```
printf_filtered ("Breakpoint %d", b->number);

if (b->source_file)
  printf_filtered ("at : file %s, line %d.",
                   b->source_file, b->line_number);
```

This is translated in the libgdb mode to (uiout is the ui_out wrapper for a stream):

```
ui_out_text (uiout, "Breakpoint ");
ui_out_list_begin (uiout, "bkpt");
ui_out_field_int (uiout, "number", b->number);
if (b->source_file
  {
    ui_out_text (uiout, ": file ");
    ui_out_field_string (uiout, "file",
                         b->source_file);
    ui_out_text (uiout, ", line ");
    ui_out_field_int (uiout, "line", b->line_number);
    ui_out_text (uiout, ".");
  }
  ui_out_list_end (uiout);
```

This example shows several elements of the libgdb interface. `ui_out_text` is a routine that outputs "human readability" data - a scripting interface should ignore this. This allows us to exactly reproduce the current gdb output, while not burdening the parser on the other end. libgdb also has the notion of keyed lists, much like the TclX keyed list facility. The lists themselves are also labeled. In this example `ui_out_list_begin` begins the accumulation of the list called bpkt, and the `ui_out_field_*` calls add fields to the list. This makes the output self-descriptive, which in turn makes the clients robust, since they are no longer dependent on output order.

There are a number of other nice features in the Libgdb interface. It supports building tables. It also has the notion of named streams, so you can accumulate to a named stream, and if some other output is generated in some function that you have called, it will go to the default output, so it won't get lost, but it will not pollute the result you are currently accumulating. The desirable property of this solution, for GDB, is that the code does not need to be inverted in order to use it from Tcl. All the changes are local, and can be made with high confidence. Given the complexity of GDB, as well as the age of the code, this is a very desirable property.

This code is not yet finalized, and except in a sandbox has not been incorporated into GDBTk yet. This will happen over the next six months, maybe even by the time you read this. We will also use this work to build the generic scripting interface to GDB, and we will implement a Tcl instantiation, while the official FSF version will probably use Guile.

Blocking Behavior:

This problem critically effects the look and feel of the GDBTk. It is the reason why almost all the other GDB GUI's chose to run in a separate process. It is important that the UI repaint itself, and that it continue to be interruptible, no matter what the target is doing. So you need to always be listening to events coming from the connection to the X-Server.

Solution 1: *Timers*

Our first solution to the problem was to use signals to wake up the GUI while gdb was blocked. There are a number of ways you can do this. One is to use SIGIO on the connection to the X Server. However, SIGIO is not delivered at all on Linux, and not reliably on other systems. So in the end we had to resort to using timers. Before each call into GDB that we knew was going to block, we would start up a timer, and then refresh the GUI in the SIGALRM handler.

This method requires some care, since you intend to run Tcl code in the SIGALRM handler. Remember, the point is JUST to refresh the interface, and not to interrupt the call that is currently blocking. You need to make sure that you never get the SIGALRM when you are not blocking, since this will lead to random crashes. This means that the time-out has to be fairly coarse-grained, and started and stopped fairly close to the place where you will block.

Although this is the method that GDBTk currently employs, it is highly unsatisfactory. It is too easy to miss a place where gdb can block. This is particularly a problem when something unexpected happens, like you lose connection to a remote board, and then a query that normally takes no time, and occurs too far down the call chain for the timers to be conveniently set and unset, suddenly blocks for the length of the "connection lost" timeout, which is generally ~5 seconds. It also causes odd occasional corruption if the timer happens to fire just on the way out of a call. The lesson we learned from this is that while with a lot of work, you can manage to fake an essentially blocking application to look like an event driven one, in the long run you will lose too often to make the fake convincing.

Solution 2: *A Real Asynchronous Event Loop*

Ultimately, we decided that there was no way to get GDB to behave well when driven by a GUI, unless it really was a true event driven application. Furthermore, there is quite a bit of information that it is legitimate to access while the target is running. You might want to browse the breakpoint list, or list files, or search for functions in the executable. So there was a good deal of motivation to make this switch, even though it involved reworking a large chunk of gdb.

At this point there were two models that we could follow. One was to make gdb a multi-threaded application, and the other was to treat the inputs that GDB had as event sources, and write a select-based event loop on the Tcl model.

Most of the inputs to gdb are, in fact, standard Unix event sources. Gdb can talk to remote targets either through the serial port, or over TCP/IP. The most common UNIX debugging interface is the procfs file system which is selectable. The other common UNIX debugging interface is ptrace. In this case, the calls can be made in a non-blocking mode, and then you get a SIGCHLD when the inferior process returns control to the debugger.

All this fit quite naturally in the Tcl event model. This is a well tested model, and we were confident that we could handle all the many platforms GDB runs on with this solution, whereas we had less confidence that we could get a multi-threaded application to run will cross all of GDB's supported platforms. The actual event handling code in GDB is strongly influenced by the Tcl model. We could not, of course, just adopt the Tcl code for political reasons, but we used the Tcl 8.0 notifier as the basis for the new design. It was also constructed so that we could easily convert all the input channels to Tcl channels, and use the Tcl event loop for GDBTk.

This was quite a major job, since GDB like Tcl keeps a much of its own state information on the stack. Interactions with the debugee can be quite complicated, often involving many restarts in what seems to the user (and to the higher level GDB API's) like an atomic operation. The work is not entirely complete, but by the time this paper is published all the code will be in place, and GDBTk will use it.

Brief Tour of GDBTk:

The rest of this paper consists of a short tour of some of the main screens in GDBTk. In general, there were no UI effects that we wanted to achieve but could not with Tcl/Tk. We found that in tabular displays, like the memory display, using a grid of labels led to very bad performance in scrolling, resizing and moving the window, on Windows platforms. However, Jeffrey Hobbes' TkTable proved a good alternative, and performed quite well both on Windows and on our X based hosts. As I discuss below, the speed of loading the source window was an issue for a while, but we were mostly able to overcome that. There is still more work to be done, both refining the current UI, and adding useful features, but the hindrance here is simply time, and not the available tools.

Source Window

The central window in all of GDBTk is the source window (Figure 1). This contains the main menus for the application, the Toolbar for controlling program execution. The toolbar also contains buttons to access the most commonly used windows. The toolbar is actually a Windows style toolbar with buttons that raise when entered. It also has a series of Combo-boxes at the bottom that allow you to select a different file or function, and change the display mode from source to assembly or mixed source/assembly.

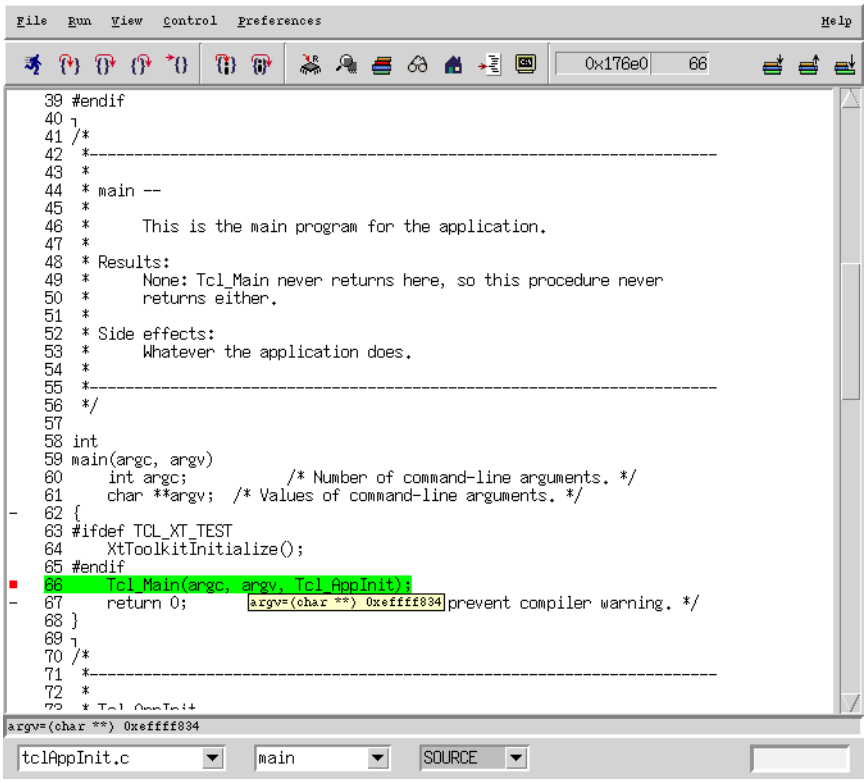

Figure 1: Source Window

Most of the work of the source window, however, is in the main text display area. The text in this display is divided by Tk Text widget tags into the break region, extending from the left edge to the end of the line numbers for lines that

contain executable code, and the source region from the end of the line numbers to the line ends. The break region is sensitive to mouse clicks, which are used to set and remove breakpoints. The source region supports variable popup balloons, and context sensitive menus for dumping memory around the variable pointed at, and some other functions.

The source window required some effort to get it to have an acceptable speed. Source files can be quite large, and when you are stepping around among a number of files, any delay in loading the window can be quite annoying. We solved this in two ways. First, once a window is rendered, we cache the text widget. Then stepping from one file to another after the sources have been hit once is quite quick.

We also read in the files in C, and pass the data directly to the Text widget's C level command. Since we actually don't want Tcl to do substitutions on the data, this is actually a sensible thing to do. There is also a lookup that has to be done for each line - to determine whether it is executable or not. It is about 3x faster to do all this work in C than to use Tcl procs to read in the line, determine whether it is executable, and run the "widget insert" command.

At this point, the rendering of text is barely fast enough. It is not annoying, but it would make GDBTk feel more responsive if we could get another 2x in rendering the text. I have not yet experimented with using the private text widget routines to populate the widget directly, though that is the obvious next step. We may also be able to speed up gdb's symbol table lookup to determine whether the line is executable or not.

Variable Window

The variable window (Figure 2) actually took much more work to get right that it seems at first glance. The main problem was getting data out of GDB as quickly as necessary to update the window. You need not only to get all the variable values, but you need to get the block information, so you can don't duplicate shadowed variables, and of course check for changed variables so you can color them appropriately. Since all the variables had to be updated with each step, any delay was noticeable.

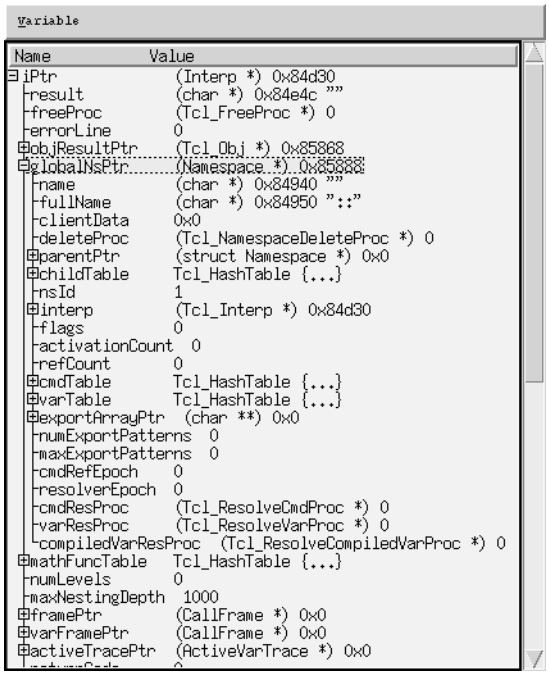

Figure 2: Variable Window

The window also allows you to change the format of variables, and to open up structures. The type and value information is not as nicely laid out as it could be, mostly because the Tix TreeWidget's multi-column layout is not as flexible as we would like. However, it is quite fast at rendering its contents, which is important.

Target Selection Dialog

The target selection dialog is particularly important to embedded developers. It is where you configure everything that is needed to get the program loaded and running on your board. Because of this its contents vary widely among the various targets and transport mechanisms. This was one area in particular where Tcl/Tk really shone. The combination of the ease with which all the elements of the dialog could be configured from a tcl array, and the dynamic rescaling allowed by the Tk geometry managers, mean that we can use a Tcl array as a flat file database for all the target specific aspects of the dialog. The implementation is very easy to read, and understand, so that external developers can quite simply tune up the settings for their particular hardware.

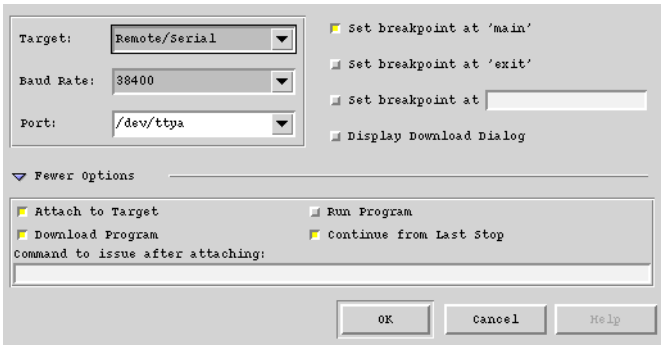

Figure 3: Target Selection

Conclusions:

Incorporating Tcl/Tk into GDB posed substantial problems. In the end, the ease and flexibility with which you can accumulate results in Tcl, and the string handling functions it provides, made it possible to overcome the lack of a real C-callable API, without requiring a massive overhaul of GDB. However, we found putting an event driven application on top of a core which felt free to block at any time, was too fragile. It was necessary to change the way GDB worked in this area, to get a really stable application.

On the Tcl side, using Itcl was a very good choice for an application of this complexity. It allowed us to segregate our program into manageable pieces, and although we did not have a very involved design, the simple inheritance that we did use was very helpful. Between the IWidgets, some widgets from Tix, and other bits from the Net, we were able to realize most of the designs that we wanted to use in GDBTk. The only real lack is a complex tree/table widget.

Acknowledgments:

Stu Grossman deserves the initial credit (or blame) for getting GDBTk started. Martin Hunt deserves the lions share of the credit for the current state of the program. Keith Seitz made many valuable contributions, and was willing to chase obscure bugs into the bowels of GDB with awe-inspiring tenacity. Stan Shebs has consistently moved GDB itself towards the new millennium, and is responsible for sheparding into being the changes to core GDB needed to put GDBTk on a firm footing. And Andrew Cagney, Fernando Nasser, and Elena Zannoni have implemented most of these new features on the GDB.