Tcl/Tk in Survivability Modeling for Military Systems

Ronald A. Bowers, Robert G. Parker, and Paul J. Tanenbaum

U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5068

rbowers@arl.army.mil, bparker@arl.army.mil, pjt@arl.army.mil

1 Introduction

One of the missions of the U.S. Army Research Laboratory (ARL) is analyzing the survivability of Army systems in the context of the entire gamut of combat hazards, from ballistic threats such as bullets, missiles, and bursting artillery rounds, through electronic- and information-warfare threats such as jamming and viruses, to biological and chemical agents. Because of this role, ARL has an active program of modeling and simulation with a history reaching more than 50 years back through a predecessor organization, the Ballistic Research Laboratory (BRL), all the way to ENIAC, the first electronic computer.

Over the past decade, as `Tcl/Tk` has developed, we have adopted it for integration into our existing applications with great success. In this paper, we discuss the problem domain, our software practices before `Tcl/Tk`, the extent and nature of our incorporation of `Tcl/Tk`, lessons that we have learned, and our plans for future efforts.

2 The Application Domain

Our earliest modeling problem was predicting how well a vehicle system could withstand a ballistic attack, or, dually, how effectively a munition could defeat a target vehicle. One major facet of this problem, clearly, is characterizing the penetration performance of whatever highly energetic stuff might be hurtled up against the vehicle skin. But another, equally important issue is determining what happens inside the vehicle in those circumstances when the threat manages to get in. The last key issue is deducing from an assortment of damaged or functional components (and people!) what critical capabilities the system retains after the attack.

Our primary model that addresses these issues is called `MUVES` (Modular `UNIX`-based Vulnerability Estimation Suite). In a typical full-scale analysis, `MUVES` must perform all these complex computations for hundreds of threats and thousands of impact locations and attack directions. Sometimes one repeats such calculations for thousands of stochastic iterations. So performance is an issue.

Of course, the effects of a munition's impact on a vehicle depend greatly on the geometric details of the weapon-target interaction, both outside and inside the vehicle. So a key input is an accurate and very-high-detail geometric description of the vehicle with all its internal components. We build and store these descriptions using a geometric modeling system called BRL-CAD (http://ftp.arl.army.mil/brlcad). In addition to editing and manipulating these descriptions, BRL-CAD also supports querying the geometry through ray tracing. This capability is packaged in an embeddable library, librt. BRL-CAD provides unmatched capability for the combination of extremely fast ray tracing of models of extremely high geometric complexity. So, again, performance is an issue.

MUVES begins an analysis by calculating the path of the threat through the target to determine which components the threat might encounter. It does this using librt. In the case of an artillery round or other fragmenting munition, there may be thousands of threats traced through the target. MUVES then calculates the penetration of every threat through the target and the damage inflicted on each component of the vehicle. Once all of the component damage has been determined, MUVES determines the target's remaining capabilities by evaluating a number of fault trees that describe the system functions in terms of components and subsystems.

Having been used for years in many Army studies and acquisition programs, MUVES and BRL-CAD now also constitute the core of the newly developed tri-Service model called AJEM (Advanced Joint Effectiveness Model), which has been adopted for use jointly by the Army, Navy, and Air Force to perform weapon system effectiveness analyses. This expanded user community brings with it new requirements for analysis and user-interface capabilities.

3 Life Before Tcl/Tk

Incorporating Tcl/Tk into large legacy software systems poses a special set of challenges. But it also offers a substantial set of benefits. We now summarize our state of affairs as they stood before Tcl/Tk.

3.1 BRL-CAD

Most users' interaction with BRL-CAD is centered around the geometry editor/viewer called mged. First developed around 1980, mged featured a command-line interface and limited mouse-based interaction with the geometry. Its purpose-built command interpreter performed globbing against database names before executing each command, much the way UNIX shells glob against file names. The program also had interactive graphical display capabilities on various platforms, most importantly a GL-based version for SGIs and an X-based version for others. The application drew its menus and scrollbars using its own special-purpose software.

We had ported mged to dozens of UNIX platforms, but it was certainly most useful on high-end graphics workstations. By the '90s the user interface was obsolescent. We had some success at modernizing the GUI with an X/Motif-based prototype called xmged. This provided buttons and menus with 3-D appearance and freed the application from the pain of drawing them. But integrating these cool widgets still required developing and maintaining a considerable amount of code.

At the same time, we began kicking around the idea of developing a VINTGUIMS (vendor-independent, network-transparent, graphical-user-interface management system). That's when we discovered Tcl/Tk, which proved to offer very nearly all the features of this ambitious idea—and many other benefits besides.

3.2 MUVES

MUVES has been in continuous development since 1986, and its design reflects that period. It includes substantial code for memory management, input parsing, and associating character-string keys with data. It also includes its own small command language for specifying patterns of shots to be fired against targets. And it is rife with global variables.

Although the MUVES ray tracer (based on librt) operated in parallel even before Tcl/Tk, the MUVES analysis was single-threaded. So even independent threats traveling along different paths were analyzed sequentially. This constituted a severe performance bottleneck.

Originally, MUVES had a termcap-based ASCII interface. We later developed a GUI and several graphical pre- and post-processors. All of these graphical tools were built in X and the Athena widget set, and few of today's MUVES developers know their inner workings.

By the late '90s, the users of MUVES were frustrated with the difficulty of setting up their runs and interpreting the results, and the developers were frustrated at maintaining the code. This prompted us to build on the success we were having in incorporating Tcl/Tk into BRL-CAD. We chose to do the same for MUVES.

4 Integrating Tcl/Tk into Our Applications

Our first step in adopting Tcl/Tk was incorporating it into mged. As we came to see its benefits, we applied it also to many of the key C libraries in BRL-CAD. Complete converts by then, we next undertook a similar Tcl/Tk overhaul of MUVES. Although we had had substantial success with plain Tcl/Tk, the size of our applications and the benefits of the object-oriented paradigm led us to adopt [incr Tcl]/[incr Tk] in its turn.

4.1 The Test Case: mged

Since it is a powerful and complex interactive tool with stringent performance requirements, mged was the ideal first candidate for Tcl-ification. We began by retiring mged's single-purpose command interpreter in favor of a Tcl interpreter. We also modified our main loop, having it make calls to Tcl_DoOneEvent. We Tcl-wrapped each mged command and registered it with the interpreter twice: once under the same name used in mged and once more with the prefix _mged_ prepended. This allows developers and users to override in Tcl the behavior of the original mged command, by defining a proc with the same name, while retaining access to the original behavior. Such an overriding proc may do some additional processing before or after calling the _mged_ command, or it may do something unrelated. For example, the mged command called ls lists all the objects in the database. By default, it creates a short listing. But users can easily establish a new default—making `ls` produce a long listing instead—by defining a proc named `ls` in their `.mgedrc` file as follows:

```
proc ls {args} {
    if {[string match {-[aclrc]} [lindex $args 0]]} {
        # at least one valid option was provided
        return [eval _mged_ls $args]
    } else {
        # no valid options found, try a long listing
        return [eval _mged_ls -l $args]
    }
}
```

The original `ls` behavior can be restored by redefining the `ls` proc as follows:

```
proc ls {args} {
    return [eval _mged_ls $args]
}
```

Our next step was to build a brand-new `Tk`-based GUI. While doing this, we `Tcl`-wrapped further scraps of our `C` code to create the callbacks for handling mouse and keyboard events. `Tk` handles all the event parsing nicely, simply executing the code registered for the appropriate events.

Because both the `C` and `Tcl` APIs for `Tcl/Tk` were so easy to use, we were able to provide much more functionality than ever before, including an extensive menu hierarchy, multiple views of the geometry, and numerous control panels that extend the main GUI. With the power of `Tcl`, `mged` is now more than a geometry editor. It has become an extensible environment in which programmers (and even mere `Tcl`-savvy users) can develop tools for tasks such as procedural creation of geometry and extension or replacement of the GUI.

4.2 BRL-CAD Libraries

While the overhaul of `mged` was under way, we also began writing `Tcl` wrappers to register many of the routines in our libraries. We now have access within `Tcl` to `BRL-CAD` library routines supporting such things as geometry creation and modification; ray tracing; manipulation of vectors, matrices, and quaternions; control of device-independent networked frame buffers; and control of display managers for `OpenGL`, `X`, `PostScript`, and other systems.

We also created "`Tcl` objects" (not to be confused with `Tcl_Obj`'s) for a number of `BRL-CAD` entities that encapsulate the attributes and methods of those entities. Instancing such a `BRL-CAD` `Tcl` object creates a `Tcl` command by the same name. The object may be manipulated through its command in much the same way as the buttons, labels, and other features of `Tk`. As an example, one can create a *database* object that supports querying and modifying the geometry in a database file as follows:[1]

[1]This example and several others illustrate the use of a program called `btclsh`, which is described below.

```
btclsh> wdb_open db scene.g
db
btclsh>
```

The resulting object provides access to the database `scene.g`. For instance, one can list the file's contents:

```
btclsh> db ls
LIGHT         cone.r/R      light.r/R     tor.r/R
all.g/        cone.s        platform.r/R
box.r/R       ellipse.r/R   platform.s
box.s         ellipse.s     tor
btclsh>
```

Inspired by our success with these `Tcl` objects, we promoted some of the code from `mged` into `librt`, both to flesh out the `Tcl` database object and to provide the functionality for additional objects. Besides databases, the currently implemented `BRL-CAD` `Tcl` objects include *display managers* for rendering geometry, *drawable-geometries* for expressing geometry in a form amenable for the display manager, and *views* for controlling a viewing transformation.

After we discovered `[incr Tcl]`, we began wrapping `BRL-CAD`'s `Tcl` objects as `[incr Tcl]` classes. The database object, for example, has been wrapped as `class Db`. Its constructor uses `wdb_open` to create a database `Tcl` object, and its methods simply invoke those of the embedded database as follows:

```
itcl::class Db {
    protected variable db ""
    public variable dbfile ""

    constructor {filename} {
        set dbfile $filename
        set db [subst $this]_db
        wdb_open $db $dbfile
    }
    destructor {} {
        rename $db ""
    }

    public method ls {args}
        ...
}

itcl::body Db::ls {args} {
    eval $db ls $args
}
    ...
```

This simple sort of wrapping allows us to make extremely productive use of inheritance. As an example of the power that this approach provides, consider our [incr Tcl] class called Mged. At present, it boasts about half the functionality of mged itself. And since this functionality is still implemented in C, there is no loss of performance. It uses many of the [incr Tcl]-wrapped BRL-CAD Tcl objects in one form or another, including display managers, views, databases, and drawable geometries. Some of the functionality is obtained through inheritance and some through embedding. This mega-widget provides users of applications built on it an ability to view, query, and modify geometry that is entirely consistent with their years of experience with the rest of BRL-CAD.

If [incr Tcl] is eventually incorporated into core Tcl, then we will replace our Tcl objects with code written specifically for methods in an [incr Tcl] class. Until then, the extra layer is a minor inconvenience.

Two new applications arose as a result of our integration of Tcl/Tk into BRL-CAD. One is btclsh, a tclsh-like convenience application that links in [incr Tcl] and the BRL-CAD libraries. It also provides command line editing like that found in tcsh using C code borrowed from mged. The other is bwish, a wish-like application that does everything btclsh does and links in Tk, [incr Tk] and the iwidgets. These two applications provide a convenient platform for developing BRL-CAD applications.

4.3 HLA Demonstrator

We have also used Tcl in distributed interoperable simulations built on the DoD's High Level Architecture (HLA), a framework in which remote independent simulations can interact with each other. One example is a terrain server that uses BRL-CAD to store terrain models and support queries. It allows remote federates such as vehicle-suspension simulations to query the terrain via HLA. A sample simulation, for instance, includes two federates: the terrain server and a driver. A man-in-the-loop interface built on mged interacts with the driver via Tcl, sending requests such as, "I am at location (x, y, z); at what altitude is the ground?" The terrain server uses librt to fire a ray downward from the specified location, returning a point on the terrain. The driver then generates a Tcl command that it sends to mged to move the vehicle into the proper position and orientation. This keeps the vehicle on the ground while preventing burrowing.

4.4 Distributed MUVES

As we added more and more detail to MUVES, it became clear that performance had become a critical issue. Runs were taking days, and in some cases weeks. One possible solution was to convert MUVES into a multi-threaded program that could take full advantage of the large multiprocessor computers available at ARL. This approach was attempted, but failed: MUVES was too complicated to be converted easily. Making MUVES multi-threaded would require a complete rewrite, and ARL management considered that too large an effort. So we had to find a solution that would yield a large performance increase without a long development time or huge manpower requirement.

The solution that we hit upon was simple. Rather than parallelize MUVES, we could simply run several instances simultaneously, possibly on multiple computers, and tie them all together. We called the new architecture *distributed* MUVES, or DMUVES. Of course, a design such as this raises many implementation issues. Fortunately for us, though, Tcl was there to help.

DMUVES uses a "boss-and-workers" parallel architecture. The boss divides the MUVES analysis into a series of independent tasks. It sends these tasks to the workers, which perform the tasks and return the results to the boss. The boss then sorts the results and prints them to a file in the order in which the previous versions of MUVES would have recorded them.

The boss and the workers communicate through sockets. Setting up the connections is greatly simplified by the Tcl socket functions. At the beginning of a run, the boss uses Tcl_OpenTcpServer to establish a rendezvous port. Each worker then uses Tcl_OpenTcpClient to attach to the boss, and thereafter talks to the boss through its own Tcl_Channel.

In order to achieve the highest performance possible, DMUVES uses the Tcl C libraries directly rather than a script. But the messages passed between the boss and the workers are Tcl scripts that are evaluated by their recipients. This allows for easy expansion of the capabilities of both boss and workers. Note that a safe interpreter is used to avoid the security risk of executing arbitrary messages.

The activity of the boss and workers is coordinated by the Tcl event-handler system. At the heart of both the boss and the workers are Tcl_DoOneEvent loops, which monitor the state of the sockets. When a readable event is detected, the corresponding message is read from the socket and stored in a queue. The Tcl_DoWhenIdle function is then invoked to set a callback for when there are no pending socket events. The Tcl_DoWhenIdle instructs the program to de-queue the next message and evaluate it. Thus, as long as there are messages inbound on a socket, the recipient will read them, and at most one task will be performed before the program returns to test the state of the socket and read any new messages.

One of the many benefits of using Tcl interpreters in the workers is that we can now load procs into the workers that do things not possible in previous versions of MUVES. For example, a proc can be used to iterate an analysis until a certain condition is met, such as convergence. As a result, Tcl enables us to adapt MUVES more quickly to our ever-changing analysis needs.

In addition to the boss and workers, the DMUVES system also includes daemons that run on each remote host that is to be used. Written entirely in [incr Tcl], the daemons start workers on the remote hosts, manage the allocation of workers to bosses, and ensure that the hosts' copies of the MUVES inputs are current.

The DMUVES design has proven quite effective and scalable. Performance gain is nearly linear and the boss has been shown capable of handling at least 96 workers. Thus, given access to enough processors, analysts can expect to get their jobs done in a small fraction of the time required by the old design.

4.5 Combining the strengths of BRL-CAD and MUVES

From its inception, MUVES has been built on BRL-CAD, and its ray-tracing capabilities in particular, but adopting Tcl/Tk has allowed developers of each package to leverage capabilities of the other to great advantage. One example is fr_viewer, a post-processing tool for investigating MUVES final results files using the visualization and manipulation capabilities of mged. This application exposes the correspondence between MUVES components/systems and BRL-CAD geometry. In the past, an analyst who needed to view or modify the geometry had to infer this correspondence through painful study of the MUVES input files.

The fr_viewer is implemented as a Tcl script that is run within bwish. It uses the iwidgets and a few mega-widgets from BRL-CAD and MUVES. Now the user can effortlessly visualize geometry in both wire-frame and ray-trace-rendered form. And the tool was quite easy to implement.

We are also developing a number of input-preparation tools for MUVES. One is an [incr Tcl]/[incr Tk] script to support interactive graphical creation of the fault trees and associated data tables that describe the functionality of vehicle systems. This is another facet of the analyst's job that has always required awkward hopping back and forth among the BRL-CAD database, vehicle engineers' descriptions of system behavior, and the MUVES input files. When completed, this tool will provide a common interface for the engineers who describe the system and the analysts who run MUVES and interpret the results.

4.6 The AJEM GUI

The existing X-based GUI for MUVES, unpopular with users and difficult to maintain, was deemed unacceptable for use by the AJEM community. So an AJEM-specific GUI was written using a proprietary X widget package. This GUI was not well received either. ARL suggested that a new AJEM GUI be written in Tcl/Tk so that it could be easily integrated with BRL-CAD and the evolving Tcl/Tk-based MUVES tools. Unlike the previous false starts, this Tcl/Tk GUI was so easy to develop that an early version was ready to ship with the first official release of AJEM.

Repeating the experience we had with the mged GUI, the AJEM developers have been able to produce a steady stream of powerful and flexible new capabilities, taking full advantage of [incr Tcl]/[incr Tk] to slash the time and effort required for the process.

5 Lessons Learned

We have always tried to write code with an eye toward reusability, but adopting Tcl/Tk and [incr Tcl]/[incr Tk] has heightened our sensitivity substantially. We continually discover new possibilities and imagine novel tools that can share functionality once thought to be specific to another application. And we see both the BRL-CAD libraries, which had offered only their C interface, and the internals of mged being transformed into a BRL-CAD toolkit that can be used by any Tcl application.

We've found GUI development with Tcl/Tk to be far easier than with other packages, including Athena, Motif, MFC (Microsoft Foundation Classes), Java's Swing, and most certainly raw X.

The most important lesson that we've learned the hard way is that when you use Tcl for large applications, variable names can quickly get out of hand. But [incr Tcl] can alleviate this, and we strongly endorse it. Another is that it's preferable, when passing interprocess messages containing floating point numbers, to transmit the data in a single binary object, rather than as an ASCII string.

One final recommendation: there is excellent training in `Tcl/Tk` available. Go forth and get some early on.

6 Future Efforts

We will certainly continue `Tcl`-wrapping `BRL-CAD` library routines and fleshing out and creating new `BRL-CAD Tcl` objects. We will also migrate more code from `mged` into the libraries. When all is said and done, the amount of `mged`-specific code should be reduced by nearly half. We also plan to reimplement the `mged` GUI, currently in `Tcl/Tk`, in `[incr Tcl]/[incr Tk]`. This will undoubtedly result in the creation of more `BRL-CAD` mega-widgets that can be used by other applications.

Another area for exploration in `BRL-CAD` is *object-oriented geometry*. The next version of the `BRL-CAD` database file format will allow each item in a database to be given arbitrary string-valued attributes. One possible use for this feature is to assign `Tcl` procs as the values for agreed-upon "method" attributes. Before this approach can be adopted, though, we must study the security implications of embedding behavior code in our geometry. Clearly, safe interpreters would be key in any implementation.

Because we began incorporating `Tcl/Tk` into `MUVES` more recently, there remain an even larger number of unexplored possibilities here than with `BRL-CAD`. These possibilities can be grouped into several broad categories:

- wrapping `MUVES` functions,
- implementing further `DMUVES` boss and worker tasks,
- coupling `MUVES` parameters and `Tcl` variables,
- building additional graphical tools, and
- replacing `MUVES` functions with `Tcl C` library calls.

Although some `MUVES` functions have been wrapped for use through the interpreter, most have not. Wrapping `MUVES` functions is hard and not very much fun for several reasons. First, most of the interesting functions rely on the pool of global variables. Properly configuring the global state is laborious. Also, most of the parameters for `MUVES` functions are of `MUVES`-specific data types, so setting them up is another chore. But performing this drudgery to wrap `MUVES` functions will be necessary in order to further integrate `Tcl` into `MUVES`.

Expressing the boss-worker messages of `DMUVES` as `Tcl` scripts allows us to easily add commands—and hence functionality—to the system. Several such commands are being considered currently. One would proceed through an analysis only to the point at which all the damage has been calculated and would return the list of damaged components. This would allow us to divide the calculation of damage among several workers. Other commands would then collect the lists of damaged components and evaluate the final state of the vehicle.

Another new command would allow a worker to "subcontract" some of its work. For example, a worker tasked with analyzing the effects of an artillery salvo could divide that task into several subtasks, each corresponding to analyzing a single artillery round. The worker could then submit those subtasks back to the boss to be farmed out to other workers.

The expression of worker commands as `Tcl` scripts has also made possible interactive development of analysis algorithms through the `Tcl` interpreter. This will allow `MUVES` developers and the physicists and engineers who develop the underlying `MUVES` methodology to write and test prototype software much more quickly and easily.

At the moment, most `MUVES` parameters are set in the input files. Coupling `MUVES` parameters with `Tcl` variables will allow us to manipulate parameters while an analysis is running. This will enable us to expand the parameter space that is explored during a single run. For example, we could modify a component's material type to determine what effect it has on the target's vulnerability.

`MUVES` analysts often request new tools to help them visualize `MUVES` results. One such tool is `fr_viewer`. No doubt, additional graphical tools will be requested. The most interesting ones merge `BRL-CAD` visualization capabilities with `MUVES` results. With `Tcl` as the common command language for both `BRL-CAD` and `MUVES`, integrating `BRL-CAD` visualization functionality into `MUVES` is simple.

Ultimately, we hope to replace much of the legacy `MUVES` code with routines from the `Tcl` `C` libraries—it is folly for an application that relies heavily on `Tcl` to include reimplementations of much of the functionality existing there. Some of this has already been accomplished. We have replaced the special-purpose `MUVES` system for associating string keys with data by a `Tcl` hash table system. However, the `Tcl`-ification of `MUVES` has only just begun. There remains much to be done.

7 Conclusion

We have incorporated `Tcl/Tk` into two critical, high-visibility applications within the U.S. Army. The result has been code that is easier to maintain and augment and that meets our very demanding performance requirements. Like any application of the toolkit philosophy familiar to `UNIX` programmers, our adoption of `Tcl/Tk` is certain to make possible the development of many highly useful capabilities as yet unknown. This, in turn, will allow the Army to develop materiel at a savings of time, tax dollars, and lives.