The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

Creating a Multimedia Extension for Tcl Using the Java Media Framework

Moses DeJong, Brian Bailey, and Joseph A. Konstan
University of Minnesota

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Creating A Multimedia Extension for Tcl Using the Java Media Framework

Moses DeJong, Brian Bailey, and Joseph A. Konstan
University of Minnesota
Computer Science and Engineering Department
{dejong, bailey, konstan}@cs.umn.edu

Abstract

As multimedia capable computers become cheaper and more pervasive in the consumer and corporate markets, and as the availability of digital information increases, the need for low-cost, cross-platform multimedia applications will steadily rise. However, because Tcl lacks native support for continuous media streams, such as audio, video, and animation, it is not well suited for this emerging application domain. At the same time, Java now provides a set of class libraries, called the Java Media Framework (JMF), which provides the multimedia support that Tcl lacks. With the recently introduced integration of Tcl and Java, Java can now be used to provide the cross-platform multimedia support required by Tcl; whereas Tcl can be used to provide the easy-to-use programming environment required for building multimedia applications. In this paper, we introduce a Tcl extension that provides a high-level scripting interface to the Java Media Framework. In addition, we will highlight some interesting problems in the current Tcl/Java package as well as suggest some potential solutions. This paper will benefit Tcl programmers who would like to learn more about using Tcl to build multimedia applications, integrating Tcl and Java, or the multimedia support provided by the JMF.

Keywords

Multimedia, Tcl extension, Synchronization, Jacl, TclBlend, Java, Java Media Framework

1 Introduction

Tcl [9] has never directly supported continuous media streams, such as audio, video, or animations. Tcl programmers wanting to include these types of media objects in order to create desktop conferencing applications, electronic manuals, voice activated interfaces, or even simple multimedia presentations, have had to rely on fellow Tcl developers to build, distribute, and support multimedia extensions. Due to the huge variety of media formats, transport protocols, playout devices, and synchronization models in use today, it is no wonder the core Tcl group has shied away from attempting to directly provide multimedia support. One of the more popular multimedia extensions for Tcl is the Continuous Media Toolkit (CMT) [10] developed at California-Berkeley by the Plateau project group. CMT is a flexible, low-level multimedia toolkit supporting a variety of media formats, transport protocols, and playout devices. Although CMT has proven to be very useful in our research, it suffers from the same problems as other Tcl extensions:

- *Portability*. Designing extensions that are portable across multiple flavors of Unix, Windows, and the Macintosh is a non-trivial task. In fact, most extensions available today, including CMT, do not provide this level of cross-platform portability.
- *Compatibility*. With each new release of the Tcl/Tk core, an extension developer must update and test their extension to ensure that it still functions properly. Unfortunately, this has not been easy as Tcl/Tk has changed substantially in at least the following core areas; file I/O, image support, sockets, event handling, and object APIs, to name just a few. For some extensions, such as CMT, this has caused a lag of at least 6 months behind new releases of Tcl/Tk.
- *Configurability*. Extensions cannot easily access the configuration information Tcl uses for its own compilation. Extension writers must effectively recreate this same information, which is why almost every Tcl extension comes with its own set of configuration scripts.
- *Dependencies*. Many Tcl extensions are not standalone; i.e., they require other extensions in order to function properly. For example, CMT requires the Tcl-Dp [12] extension in order to leverage distributed services. As the number of extension dependencies increases, so does the difficulty of upgrading an extension to support the latest release of Tcl/Tk.

Within the last year, JavaSoft has introduced the Java Media Framework (JMF) [4]. The JMF is a Java [1] package that provides support for the local playback of a wide variety of audio (AIFF, AU, DVI, MIDI, and MPEG-1), video (H.261, H.263, MJPEG, MPEG-1, AVI, QuickTime), and animation formats (Apple Animation). As the JMF evolves, it promises to also support audio recording, audio mixing, video capture, streamed playback, and emerging multimedia standards such as MPEG-4. The JMF already provides roughly the same functionality offered by CMT, but without many of the problems inherent in Tcl extensions as described above. Thus, the JMF deserves serious consideration as the underlying toolkit used in commercial multimedia applications or even research projects. However, building multimedia applications in pure Java code using the JMF has several drawbacks:

- *Complexity*. The JMF currently supports only primitive media events such as notification of a playback rate change. Exactly how these events affect the rest of the application must be defined by the developer in low level Java code.
- *Lack of higher-level synchronization models*. Without higher-level synchronization support, building complex multimedia applications becomes difficult. This is roughly equivalent to building user interfaces by invoking low level X routines instead of using a higher-level toolkit like Tk.
- *Edit/Compile/Debug cycle*. Strongly typed, compiled languages like Java are great for developing low level systems, but they make poor tools for rapid prototyping and development of high-level multimedia applications.

Because of these drawbacks, we argue that the JMF would benefit from a scripting interface that would increase the usability and decrease the development effort required to build complex multimedia applications. Thus, the goal of this project was to use the Tcl/Java package [5, 6, 13] to provide a scripting interface to the JMF called **TJMF** (**T**cl/**J**ava **M**edia **F**ramework). This paper describes our experiences and lessons learned from building this Tcl interface to the JMF. In section 2, a general overview of the features already provided by the Tcl/Java package is presented. Section 3 outlines the utility packages that provide the foundation for the TJMF extension. In Section 4, an example of using the TJMF extension to playback a MPEG movie as well as a detailed discussion of the extension is provided. In Section 5, we reflect on our experiences and lessons learned from using the Tcl/Java package. In the remaining sections of the paper, we present our conclusions and the results of recent work to improve the Tcl/Java package.

2 The Tcl/Java Package

To better integrate Tcl and Java, several key technologies have been developed. Jacl [6] is an implementation of a Tcl interpreter written entirely in the Java language. Jacl currently provides most of the functionality of the Tcl 8.0 interpreter. TclBlend [5] provides a native code library that accesses the Java Virtual Machine (JVM) using the Java Native Interface (JNI). The Java Package [5] is a set of Tcl commands providing access to the Java Reflection system. The Java Package provides commands that allow a Tcl interpreter to both allocate and invoke methods on Java objects. Tcl code using these commands will run seamlessly in either the TclBlend or Jacl implementations. In addition, extensions written in Java using the Java Package can be accessed in either Jacl or TclBlend, and are instantly portable to a wide variety of systems without configuration or recompilation. For the purposes of this paper, we use the term Tcl/Java package to collectively refer to Jacl, TclBlend, and the Java Package.

3 Integration Utilities

Before describing our multimedia extension for the Java Media Framework in detail, several supplemental packages need to be introduced. These utility packages provide the underlying support required by our multimedia extension and consist of the following:

- *Object-oriented support*. The lack of object-oriented support in Tcl makes modular development difficult. For several reasons that will be explained shortly, none of the current object-oriented extensions were usable in this project, and thus a Tcl-only object-oriented package was created.
- *Event mapping system*. A package to map Java events to Tcl callbacks was created so that events from the JMF components could be managed in user-defined Tcl code.
- *Pack geometry manager for the Java AWT*. The existing Java layout managers are inflexible and difficult to use. In response, a new layout manager, based upon the pack geometry manager of Tk, was created for Java AWT components.
- *Package command for Jacl*. Initial versions of Jacl did not provide an implementation of the Tcl

`package` command, but since our project required this functionality, the Jacl interpreter was extended with an equivalent Java implementation.

3.1 Object-Oriented Support

From our experience, the most severe impediment to modular development in Tcl is the lack of object-oriented support. Tcl has no built-in support for aggregate data types, other than lists and arrays, and also lacks support for encapsulation of data. [incr Tcl][8], an extension that provides object-oriented support, could not be used in this project for two reasons. First, [incr Tcl] requires patches to the Tcl core and its own C level libraries, and thus cannot be run in Jacl. Second, TclBlend requires the use of Tcl8.0 which is not yet supported by [incr Tcl]. Although several Tcl-only object-oriented packages exist within the Tcl community, we could find none that would seamlessly work with both Tcl8.0 and Jacl. In one case, the use of "::" within global procedure names conflicted with the new namespace feature of Tcl8.0. Because of these problems, we designed a simple, Tcl-only, object-oriented package that is compatible with all current Tcl releases, including Jacl. The package provides basic support for classes, encapsulation, and composition.

3.2 Event Mapping System

In order to script JMF components using Tcl, Java events need to be mapped to Tcl callback procedures. This type of mapping is similar to the way X events are mapped to Tcl procedures using the `bind` command from Tk. Such functionality would allow a Tcl programmer to create JMF components; e.g., a MPEG player along with a set of VCR controls, and manage the component interaction in Tcl code. The `java::bind` command included in the Tcl/Java package already provides this type of Java to Tcl event mapping, but could not be used because it only supports JavaBean events, which the JMF does not generate. In our TJMF extension, a Java class receives JMF events, translates them into a descriptive string, and invokes the appropriate Tcl callback. Example 1 demonstrates how a Tcl callback would be invoked when a Java event is received from the JMF.

3.3 Pack Geometry Manager

Java's Abstract Windowing Toolkit (AWT) provides poor support for geometry management. The default layout managers are lacking in functionality, error prone, and difficult to use. On the other hand, Tk's pack geometry manager is powerful, easy to use, and allows for rapid prototyping of user interfaces. In order to gain these same advantages when developing interfaces with the AWT, a pack geometry manager was created by porting the packer layout algorithm to the AWT 1.1 LayoutManager interface. Java programmers can use this layout manager to gain pack functionality for AWT components. Although this functionality could be accessed from Tcl using the Tcl/Java package, the layout manager would be easier to use if wrapped in a Tcl command. For this reason, a `jtk::pack` command was created that accepts the same options as the `pack` command of Tk. Use of this command can be seen in Example 1.

3.4 Package Command

Tcl's `package` command provides a convenient means for managing logically connected source files within an application. Without it, managing which source files have already been loaded into the interpreter can become very complex. Unfortunately, early versions of Jacl did not support the `package` command. Initially, the Tcl/Java group was contacted, but since they could not begin working on unimplemented features until after the 1.0 release date, the decision was made to implement the `package` command ourselves. The implementation was performed by translating the source code from the C version of the `package` command into Java source code. The Java version of the command was then checked by running the regression tests from the main Tcl distribution. Once stabilized, the code was sent to the Tcl/Java group and has since been included in subsequent releases of Jacl.

These packages provide the foundation on which the TJMF extension is built. With each of these packages in place, a detailed explanation of the TJMF extension will now be provided.

4 The TJMF Extension

The motivation behind the TJMF extension began while assisting another research project in the University of Minnesota's kinesiology department. The kinesiology department was conducting research involving a sports performance measuring system that required a multimedia interface. While a number of multimedia technologies are available, the decision was made to use Java and the JMF. Writing Java code requires much less time than a more complex language like C or C++, but development was still slow due to the edit/compile/debug cycle; especially because restarting the application from scratch took a long time. Although the project was a success, the implementation effort could have been reduced by eliminating the need to continually recompile and

restart the application. Having used multimedia scripting tools in the past, it was clear that application developers using the JMF would benefit from a scripting front-end. Thus, the idea of providing a Tcl scripting interface to the JMF was conceived.

4.1 TJMF Usage

The interface to our TJMF extension was designed with two goals in mind. First, Tcl programmers should be able to use the TJMF with minimal effort. Second, Java programmers already familiar with the JMF, but who desire a faster development cycle, could use the TJMF without learning a whole new multimedia toolkit. This should help convince Java programmers using the JMF to switch to Tcl and the TJMF. Example 1 demonstrates how a MPEG file would be displayed with the TJMF extension.

Example 1

```
#Add TJMF packages to this interpreter
package require Media

#Create a MPEG player
set player \
[MediaPlayer .play file:/C:/mpegs/movie.mpg]

#Create the event consumer object
set con [Consumer .con]

#Subscribe to events produced by MPEG player
$con subscribe $player

#Register an event handler
$con handler Media.PrefetchComplete \
  ready_callback

#Allocate Java Frame (toplevel in Tk)
set frame [java::new java.awt.Frame]
$frame setSize 300 300

#Callback for Media.PrefetchComplete event
proc ready_callback { player } {
  global frame

  #Get window that MPEG will be displayed in
  set child [$player getVisual]

  #Pack MPEG video window in the Frame
  jtk::pack $child -in $frame -padx 10

  #Map the Frame object
  $frame show

  #Start playback of the MPEG video
  $player start
}

#Start fetching MPEG data
#Generates Media.PrefetchComplete when done
$player prefetch
```

4.2 MediaPlayer Implementation

The majority of the TJMF functionality is contained within the MediaPlayer class. The MediaPlayer class can be used to playback a number of different media types such as audio, video, and animation. This class is implemented with our object-oriented package and provides an API that programmers can use to manipulate the underlying media streams. The MediaPlayer class encapsulates three key components that work together to provide the functionality of the TJMF extension.

The first component is an instance of the JMF class `javax.media.Player`. An instance of this class is created by invoking the Java method `javax.media.Manager.createPlayer()` and passing in the URL argument supplied to the MediaPlayer constructor. The MediaPlayer constructor returns a command procedure used to access that particular instance. When this command is invoked, the MediaPlayer translates it into the corresponding method invocation on the `javax.media.Player` object.

The second component encapsulated by the MediaPlayer is a Java class called an EventTranslator. The EventTranslator class implements the `javax.media.ControllerListener` interface so that it is able to receive events generated by the JMF. The EventTranslator is responsible for receiving a JMF event, and then translating it into a Tcl string that represents the event. For example, if the JMF generates a `javax.media.StopEvent` then the EventTranslator object will output the string `Media.Stop`. This output string is then used by the third component to generate a Tcl event.

The third component of the MediaPlayer is an instance of the Producer class which was implemented in Tcl code with our object-oriented package. When the JMF generates an event, the MediaPlayer uses the output of the EventTranslator as the input to the Producer object. The Producer contains a table that uses the event string to lookup a list of Consumer objects registered for that particular event. Each consumer registered for the event will then invoke its own user-defined callback.

Together, these three components make up the MediaPlayer implementation and define the process by which JMF events trigger user-defined Tcl callbacks.

4.3 Designing a Thread Safe Extension

When implementing the TJMF extension, one issue that needed to be addressed was that the JMF creates

an additional thread to fetch, decode, and display the specified media stream. This separate thread generates asynchronous state update events which are received by the EventTanslator of the MediaPlayer (see section 4.2). After the event is translated, a procedure will need to be evaluated within the Tcl interpreter. However, because the Tcl interpreter is executing in a different thread, calling `interp.eval()` from the JMF thread could corrupt the interpreter. The problem is that the `interp.eval()` method is not thread-safe. The only safe way to evaluate a Tcl command from a separate thread is to place that command into a thread-safe Tcl event queue. To do this, the synchronized method `interp.getNotifier().queueEvent()` should be invoked. This method uses the synchronization primitives provided by the JVM to ensure thread-safety within the Tcl event queue. The Tcl interpreter will then remove the event object from the queue and invoke its `processEvent()` method from the interpreter's main thread. Example 2 demonstrates how to safely evaluate a procedure from a separate thread.

Example 2

```
String cmd = "string length hello";
EventProcessor ep = new EventProcessor(cmd);
interp.getNotifier().queueEvent(ep,
TCL.QUEUE_TAIL);

class EventProcessor extends TclEvent {
  private String script;
  EventProcessor(String s) {script = s;}

  public int processEvent(int flags) {
    try {
      interp.eval(script,flags);
    } catch (TclException e) {}
    return 1;
  }
}
```

4.4 Adding Higher Level Synchronization Support

Multimedia presentations are a common type of application that programmers may want to create using the TJMF extension. However, building these types of applications is not simple as several different approaches exist [2]. The Nsync toolkit [2] simplifies the creation of interactive multimedia presentations by providing a high-level, declarative interface. To increase the usability of the TJMF extension, we have produced a small compatibility layer so that the Nsync toolkit can be used on top of TJMF. By using the Nsync APIs, underlying details of the TJMF extension are hidden from the application programmer. We believe this adequately demonstrates the flexibility of the TJMF extension as scripts that were originally written for Nsync and CMT can be used with TJMF without significant modification.

5 Lessons Learned

In this section, we reflect on some of the issues faced during the development of the TJMF extension. While the Tcl/Java package has many useful features, it contains some problems that made development of the extension more difficult than it needed to be. The most serious problem in the Tcl/Java package is a design flaw that discards a Java object's type when reflected inside a Tcl interpreter. Java method invocation is also more difficult than is should be because the Java method resolver used in the Tcl/Java package is currently lacking needed functionality (see Section 5.2). In addition, problems exist when mapping some Tcl constructs to Java code and vice versa. These problem areas will be illustrated in the following subsections through a number of source code examples along with suggestions for improving the Tcl/Java package.

5.1 Java Objects Have No Class

Within the Tcl/Java package, all Java objects reflected in the Tcl interpreter lose their type information. Type information for Tcl objects is unimportant because "everything is a string"; however, this type information *is* important to the proper use of Java objects. Consider an example involving polymorphism. In Java, polymorphism is expressed through inheritance and allows a single object to be referenced as two distinct types (classes). In this situation, the type of the reference determines which methods can be invoked on the object. As shown in Example 3, the current Tcl/Java package discards the returned object's type by incorrectly casting it to the most derived type. By doing so, the user can now invoke methods that would ordinarily not be permitted.

Example 3

```
public class Hashtable2 extends
java.util.Hashtable
{
  public static java.util.Hashtable get() {
    return new Hashtable2();
  }
  public String foo() {
    return "CALLED";
  }
}

% set h [java::call Hashtable2 get]
% java::info class $h
Hashtable2
% $h foo
CALLED
```

This example demonstrates that the type information for the Java object is lost when it is reflected inside a Tcl interpreter. The type of the returned Java object should have been `java.util.Hashtable`, not `Hashtable2`. To fix this problem, the Tcl/Java reflection system must record the class (type) information for each Java object referenced. This modification would also allow the use of Java *interface* classes, which are currently inaccessible. To completely resolve this issue, a command to cast a Java object from one type to another would need to be included. We suggest adding a `java::cast` command as described in Example 4.

Example 4

```
#using Hashtable2 class from Example 3
% set h [java::call Hashtable2 get]
% java::info class $h
java.util.Hashtable
% $h foo
no such method "foo" in class java.util.
Hashtable
% set h2 [java::cast Hashtable2 $h]
% $h2 foo
CALLED
```

The behavior demonstrated in this example is consistent with common object-oriented principles.

5.2 The Java Method Resolver

In Java, *overloading* occurs when two or more methods have the same name but differ in the number and/or types of arguments. In order to distinguish among overloaded methods, a resolution algorithm must be applied to the argument types. Applying this algorithm is the responsibility of the Java method resolver.

When Tcl code makes a call to an overloaded Java method, the Java method resolver is invoked. In order to function properly, the Java method resolver must know the actual types of the arguments being passed. This information must be explicitly passed by the Tcl programmer using a special syntax. However, continually specifying method argument types quickly becomes tedious and error-prone. The Java method resolver is supposed to employ heuristics in order to disambiguate overloaded methods [6]. However, as demonstrated in Example 5, these heuristics are not employed.

Example 5

```
public class A {
  void foo(int i) {}
  void foo(String i) {}
}

% set obj [java::new A]
% $obj foo 1
ambiguous method signature "foo"
% $obj foo abcd
ambiguous method signature "foo"
```

The method resolver ignores Java object type information that could have been used to disambiguate the method signature. In fact, the reflection system already has this type information stored internally, it is simply ignored by the resolver. Example 6 demonstrates this problem.

Example 6

```
public class B {
  public void foo(java.util.Hashtable h) {}
  public void foo(java.util.Vector v) {}
}

% set hash [java::new java.util.Hashtable]
% set B [java::new B]
% $B foo $hash
ambiguous method signature "foo"
```

The only way to guarantee proper behavior by the resolver is to fully qualify each parameter of the Java method invocation. Using built-in Java objects, the Tcl programmer would have to specify object names like `java.io.File` or `java.util.zip.ZipFile`, which is annoying but still doable. If one starts working with classes that have longer names like `com.sun.java.swing.plaf.ToggleButtonUI`, the fully qualified syntax will quickly become tedious and error prone.

5.3 Parameter Specification

Within the Tcl/Java package, method invocations must be specified using the following format:

`{method_name f1,f2,...,fn} a1,a2,...,an`

where the f's represent the formal argument types and the a's represent the actual argument objects. The problem with this format is that every parameter type must be specified, regardless if that parameter type is required to resolve the method invocation. To alleviate this requirement, a more Java-like argument type specification could be used. Example 7 demonstrates an approach where only the parameter types needed to distinguish one method from another are supplied.

Example 7

```
public class C {
  public void foo (
    java.util.Hashtable h,
    java.util.Vector v,
    java.io.File f) {}

  public void foo(
    java.util.Hashtable h,
    java.util.Stack s,
    java.io.File f) {}
}

#This example assumes that h is a Hashtable
#v is a Vector, and f is a File object
% set C [java::new C]                    (a)
% $C {foo java.util.Hashtable            (b)
    java.util.Vector java.io.File} $h $v $f
% $C foo $h (java.util.Vector) $v $f     (c)
```

The second command above **(b)** shows the required method invocation format for the current Tcl/Java package. The third command above **(c)** shows the suggested notation in which only the necessary parameter types are specified.

5.4 Exceptional Problems

Exception handling is a widely used mechanism for raising, catching, and handling unexpected run-time conditions. Both Tcl and Java support similar exception handling mechanisms; however, a serious distinction exists. In Java, unlike Tcl, more than one type of exception can be thrown, and subsequently caught based on its type. In Tcl, this is not possible. As a result, handling Java exceptions in Tcl code is more difficult than it would be in Java code. Example 8 demonstrates the problem.

Example 8

```
public class D {
  public static void foo()
    throws NumberFormatException,
    java.io.IOException {
    //throw one of the exceptions
  }
}

//Handling the exceptions in Java code
public class E {
  public static void callfoo()
    throws NumberFormatException {
    try {
      D.foo();
    } catch (java.io.IOException e) {
      System.err.println(
        "Fatal : IoException");
      System.exit(-1);
    }
  }
}
```

```
#Handling the exceptions in Tcl code
if [catch {
  java::call D foo
} err] {
  if {[string match "*java.io.IOException*"
  $err]} {
    puts stderr "Fatal : IoException"
    exit -1
  }
  if {[string match
"*java.lang.NumberFormatException*" $err]} {
    global errorCode
    java::throw [lindex $errorCode 1]
  }
}
```

As shown in Example 8, Java exceptions are difficult to manage in Tcl code because Tcl inherently recognizes only a single type of exception. The situation becomes even more difficult when Tcl code needs to catch Tcl errors as well as Java exceptions. Every possible Java exception would need to be handled individually. Also the Tcl `catch` command does not provide a means to do local cleanup of resources without actually catching the exceptional condition. Java provides this functionality with the `try-catch-finally` construct. A `java::try` command would address this problem, but a better long-term solution would be to replace the Tcl `catch` and `error` commands with similar commands that are able to manage multiple exception conditions.

5.5 Combining Tk And AWT

At some point in the future, the Tcl/Java package must address the integration of the Java AWT and Tk. Currently, support for putting Tk windows inside Java windows, or vice versa, does not exist. This kind of support was described in [5], but has yet to be implemented.

5.6 Jacl

During this project, we discovered several problems with Jacl. First, run-time performance is poor. Jacl does not have a compiler like its C counterpart, so each command must be re-evaluated for every invocation, and Java code is generally about 10 times slower than C code. Second, Jacl can not currently be run inside of web browsers. The reason is that Jacl requires Java Reflection support which is not yet properly implemented in web browsers. Also, Jacl defines its own class loader which violates the default security policy for applets. Finally, Jacl contains a number of unimplemented, or only partially implemented commands, such as `socket`, `exec`, and `namespace`.

6 Conclusion

The TJMF extension is a powerful tool for adding multimedia capabilities to the Tcl programming language, without the inherent problems associated with C extensions. This was accomplished by using the Tcl/Java package to provide a scripting interface to the Java Media Framework. The TJMF extension was constructed with a liberal mix of Java and Tcl code:

Table 1: Lines of Tcl and Java code in the TJMF

Component	Tcl code	Java code
Media package	1000	500
Object package	1700	0
Event package	1300	0
Pack layout	100	1100
Other utilities	500	0
Total	**4600**	**1600**

The total implementation effort produced about 6200 lines of combined Java and Tcl code and was accomplished over a period of about 6 months. Each of the packages described in this paper are available for downloading at

http://www.cs.umn.edu/~dejong/tcl

Also, the object type reflection system outlined in section 5.1 and some of the method resolver heuristics described in section 5.2 have been implemented. These improvements will be made publicly available so that others can make better use of the Tcl/Java package.

By integrating Tcl and Java, each language can be used to do what it does best. Java can be used to provide the functional low-level components, while Tcl can be used to build applications by "gluing", or scripting these components together. The integration of Tcl and Java produces functionality that is beyond what could be achieved by using either language alone. With a few simple improvements outlined in this paper, the integration would become even more powerful and easier to use.

While much software has been created for Tcl/Tk, we have yet to see a solution to the portability problems inherent in using C as an extension language for Tcl. With the arrival of the Tcl/Java package, we hope that C extensions will become a thing of the past as the number of portable extensions for both Tcl and Java increase.

7 Acknowledgments

We are grateful for the help provided by the entire Tcl/Java team. Bryan Surles was particularly helpful in providing us with information about the proper use of the Tcl/Java package APIs. We would also like to thank Ray Johnson, Scott Stanton, Melissa Hirschl, and Ioi Lam as their hard work made possible the success of our project.

This work was supported by a grant from the National Science Foundation (IRI 94-10470).

8 References

[1] K. Arnold and J. Gosling. The Java Programming Language. *Addison-Wesley Publishing Company*, 1994.

[2] B. Bailey, J. Konstan, R. Cooley, and M. Dejong. Nsync – A Toolkit for Building Interactive Multimedia Presentations. *Proceedings ACM Multimedia*, 1998.

[3] S. Iyengar and J. Konstan. TclProp: A Data-Propagation Formula Manager for Tcl and Tk. *Proceedings of the 1995 Tcl/Tk Workshop*.

[4] http://java.sun.com/marketing/collateral/jmf.html.

[5] R. Johnson. Tcl and Java Integration. http://www.scriptics.com/java/tcljava.ps.

[6] I. Lam and B. Smith. Jacl: A Tcl Implementation in Java. *Proceedings of the 1997 Tcl/Tk Workshop*.

[7] J. Levy. A Tcl/Tk Netscape Plugin. *Proceedings of the 1996 Tcl/Tk Workshop*.

[8] M. McLennan. [incr tcl] – Object-oriented Programming in Tcl. *Proceedings of the 1993 Tcl/Tk Workshop*.

[9] J. Ousterhout. Tcl and the Tk Toolkit. *Addison-Wesley Publishing Company*, 1994.

[10] L. Rowe and B. Smith. A Continuous Media Player. *Network and Operating Systems Support for Digital Audio and Video*. *Third Int'l Workshop Proceedings*, 1992.

[11] A. Safonov. Extending Traces with OAT: an Object Attribute Trace package for Tcl/Tk. *Proceedings of the 1997 Tcl/Tk Workshop*.

[12] B. Smith, L. Rowe, and S. Yen. Tcl Distributed Programming. *Proceedings of the 1993 Tcl/Tk Workshop*.

[13] S. Stanton and K. Corey. Tcl/Java: Toward Portable Extensions. *Proceedings of the 1996 Tcl/Tk Workshop*.

[14] J. Swartz and B. Smith. A Resolution Independent Video Language. *Proceedings ACM Multimedia*, 1995.