The Tycho User Interface System

Christopher Hylands, Edward A. Lee, H. John Reekie
School of Electrical Engineering and Computer Sciences
University of California – Berkeley
Berkeley CA 94720
{cxh,eal,johnr}@eecs.berkeley.edu

Abstract

Tycho is the next-generation user-interface system we are building for the Ptolemy project. It is a complete [incr Tcl] application structured as an extensible class library. Our goal is to make it easy to extend this basic application with functionality and a user interface for specialized applications such as electronic design and simulation. The Tycho library includes a selection of general-purpose widgets, syntax-sensitive text editors, and graphical editing support. It incorporates architectural features that make it easy for different editors and viewers to share data and screen space. Finally, structured support for incorporating C and Java packages into this framework allows us to use those languages to complement the scripting and user-interface features of Tcl/Tk.

1 Introduction

Tycho [7] has grown from our frustration with the user-interface facilities of the present version of Ptolemy, a large C++ software package that is used to design, simulate, and generate signal processing and communications systems [6, 8, 1]. In the summer of 1995, we began to explore [incr Tcl]/[incr Tk] [11] (then at version 1.5) as a potential candidate for replacement of the Ptolemy user interface. As the project evolved, the focus of the project has shifted away from merely a replacement user interface, towards providing a framework within which we can integrate Ptolemy, new user-interface features, new simulation sub-systems coded in Java, and documentation for all of these.

Ptolemy, written mostly in C++, runs on a dozen flavors of Unix. It currently contains over 350,000 lines of code, and maintaining and building releases that support all of these platforms in a system this size is very resource-intensive (especially for an academic research group). And we still can't support Windows or Macintosh. [incr Tcl]/[incr Tk] (and Java), on the other hand, seem to provide an ideal opportunity to get out of the "binary-building business" and into the "platform-independent software business."

The [incr Tcl]/[incr Tk] part of Tycho is structured as a reusable and extensible framework of classes. Tycho will run in a vanilla `itkwish`, although we also have binaries for Tycho-with-Java, and Tycho-with-Ptolemy. A small number of C-code packages provide support we need in specific applications, such as access to real-time timers. Tycho has been developed on UNIX platforms and most of it works under Windows NT; the Macintosh port is functioning but unreliable.

Tycho is a complete application, not just a library of widgets. By default, Tycho starts up with a welcome window and its own Tcl shell, with menus that give access to all of its functionality (figure 1). It is, however, an *extensible* application: wherever possible, we have tried to build class hierarchies that allow developers to inherit or compose key functionality, so that they need to provide only the additional functionality needed for their application. We use Tycho as our [incr Tcl] development environment, and anticipate that on-going development of C and Java compilation support will make Tycho key infrastructure for all research in the Ptolemy group by the end of this year.

2 Background and motivation

Ptolemy is a software package that is used to design signal processing and communications systems, ranging from designing and simulating algorithms to synthesizing hardware and software, parallelizing algorithms, and prototyping real-time systems. Ptolemy was started in 1990, with version 0.7 scheduled for release in June 1997. Ptolemy has hundreds of active users in industry,

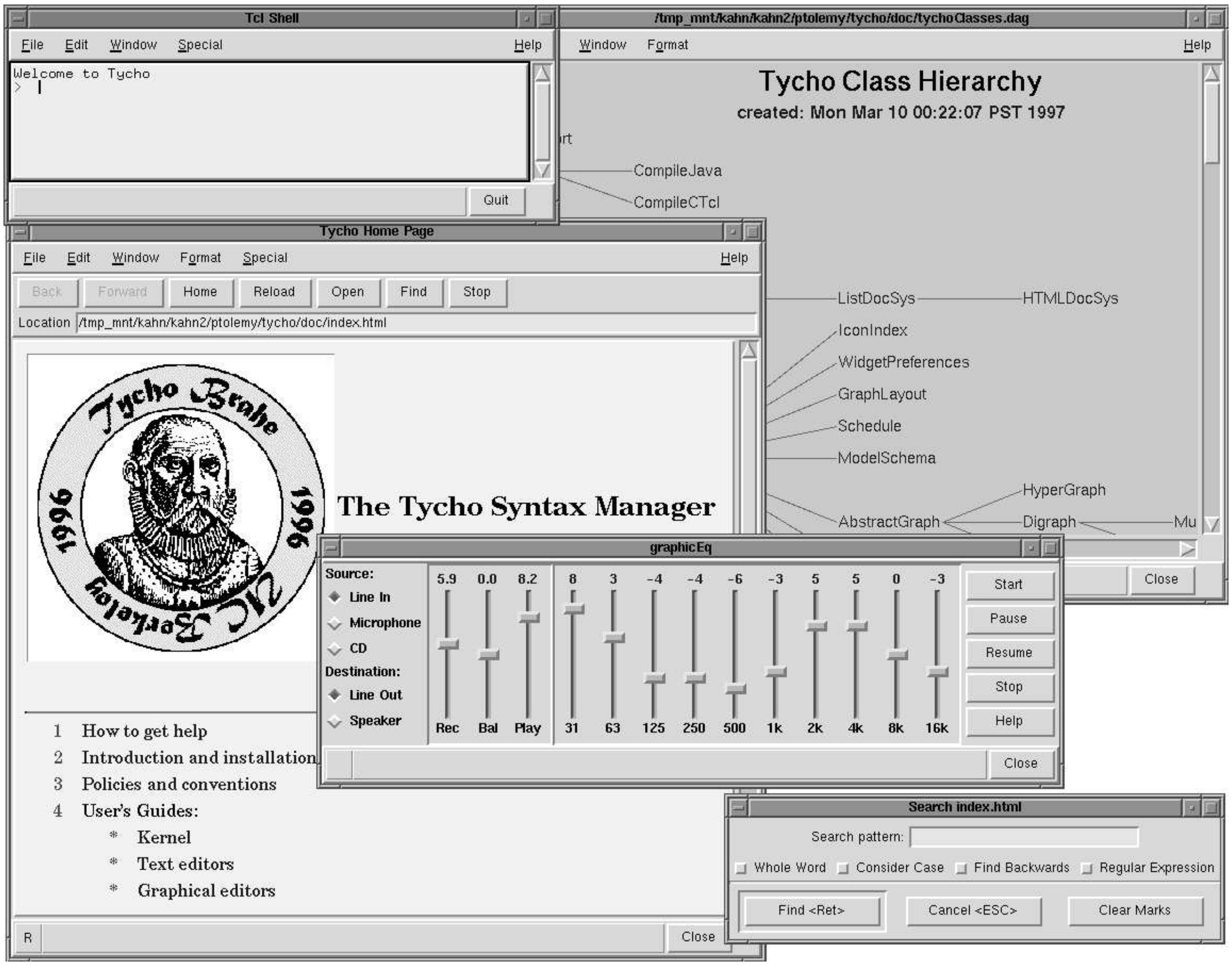

Figure 1: Tycho in action

academia, and government, and an active newsgroup (`comp.soft-sys.ptolemy`).

Ptolemy's current graphical editor is based on the VEM package developed at Berkeley in the mid-80's. Figure 2 shows the block-diagram editor. Users construct simulations by placing and connecting icons representing signal processing components on the screen. Because of its age, this interface is clumsy, hard-to-use, and out of step with modern user expectations. One of Tycho's original design goals is to replace this outdated interface, and Tcl/Tk was a clear choice for us.

Tcl/Tk was added to Ptolemy in 1992 to provide interactive and animated simulation runs. Since late 1996, Ptolemy has been running the [incr Tcl] interpreter instead of Tcl, and all new user interface work in Ptolemy is now done in the Tycho framework. (Note, however, that Tycho can also be run independently of Ptolemy.) Currently, Tycho and the older graphical editor are used together. While the older editor is still used for the block diagrams, Tycho's text editors and HTML viewer are used to browse and edit block definitions and documentation. Tycho's Tcl Shell can be opened from within Ptolemy, which provides a command-based Tcl interface to the Ptolemy kernel. We hope to complete the transition to the Tycho user interface over the summer.

One of the key factors in the success of the Ptolemy project has been its non-dogmatic support of multiple semantic models. That is, it encourages designers to produce a *domain* that encapsulates a specific model of computation and to interface it with other domains. Thus, Ptolemy supports various dataflow models, discrete-event simulation, finite-state machine models, and others. Tycho aims to provide a similar level of support for the syntactic component of system design: a designer should be able to rapidly construct a user interface to represent a particular semantic model or to vi-

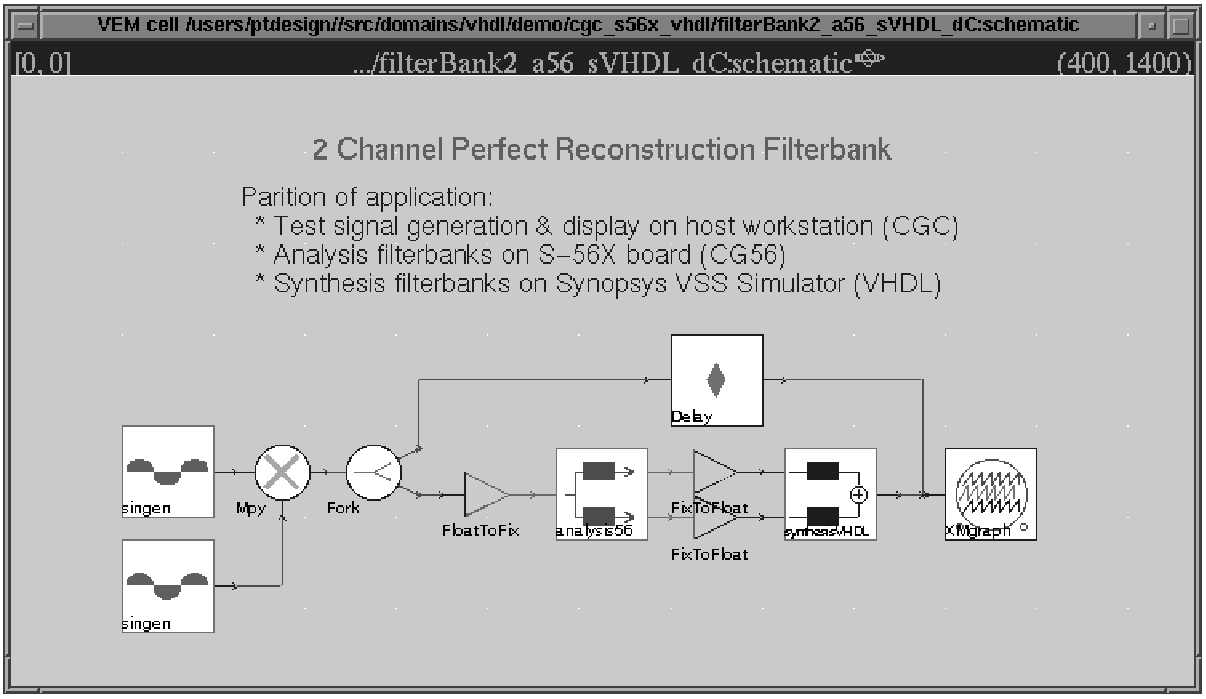

Figure 2: Ptolemy's current block diagram editor

sualize application-specific design information. For researchers in our own group, we aim to provide the support needed to rapidly construct customized user interfaces to signal processing and communications simulations. More broadly, Tycho has become an expression of our ideas about Tcl/Tk development environments: we want to provide an extensible framework in which mundane tasks such as documentation generation and indexing, font management, color management, and dialogs with the user are built using a shared, common infrastructure.

3 Tycho's development support

One of Tycho's goals is to make it easy to use Tycho itself as our development environment. To this end, we have incorporated interfaces to a few of the "most useful" development tools into Tycho. All of Tycho and Ptolemy is indexed by Glimpse [10], and this can be brought up from any Tycho window. We use SCCS revision control on all our sources, and so have a revision control dialog window that provides the most commonly-used commands, as well as the ability to regenerate any past history on demand. (Although we use SCCS, Tycho supports RCS equally well.) An off-line script processes every file in the documentation tree to produce various indexes of the documentation – every Tycho window can bring up viewers into these indexes to jump to needed documentation. A class browser (shown in figure 1) provides on overview of and access to every class in the system. Any text editor can bring up a spell-checking interface.

Tycho includes a number of syntax-sensitive text editors. We are not trying to duplicate the functionality of, say, `emacs`, but to provide a solid set of basic features which can work in concert with graphical interfaces and the integrated documentation system. Language-specific classes extend the basic editing facilities with support for compilation (C, C++ and Java), automatic documentation generation (Tcl and [incr Tcl]), and language-specific file, class, and function templates. Again, this is made possible by inheritance. Often, customizing functionality is simply a matter of inheriting from a suitable class and overriding one or two methods. For example, we have a shell that "looks inside" and monitors all activity with a *model* object (see section 4.1). This class inherits from the *TclShell* class and overrides just one method: *evalCommand*, which processes the text of an input command.

A great deal of Tycho's documentation is extracted from its source code. We use a similar scheme to Sun's Javadoc system [4], which extracts documentation from comments preceding class and method declarations, and uses special tags to distinguish different fields of the comment text. Text within comments is formatted in HTML, as is the generated documentation and indexes. All documentation can be viewed from within Tycho or using an external browser such as Netscape. In Tycho, anything can have a hyperlink to anything else – documentation to sources and vice versa; graphical editors to textual; and so on.

The Tycho HTML widget, based on Sun's HTML parsing library and shown in figure 1, has some enhancements that may be useful to other Tcl developers. First, the library relies on calls to `unknown` to handle unrecognized HTML tags. By defining all possible tags with null operations where necessary, HTML parsing sped up by a factor of three! Another performance problem was related to scoping in mega-widgets: we needed to create the text widget component at the global scope in order to prevent calls to `unknown`. Second, the HTML widget supports a `tcl` tag, which marks executable Tcl – any text so marked can be executed by double-clicking on it. This simple extension is surprisingly useful. For example, every class file contains a few lines of executable Tcl to demonstrate how the class may be used, which serves as both an example to a documentation reader and as a confidence check that documentation and code are up to date. We have taken this further where appropriate, and have written on-line tutorials for most of the user-interface support classes, which can be executed by reading and clicking.

4 Architectural patterns

4.1 The Model-View pattern

Model-view is a derivation of Smalltalk's model-view-controller (MVC) architecture; the model-view derivation combines MVC's view and controller into a single abstraction [2]. In Tycho, we have implemented a *Model* class that provides a publish-and-subscribe mechanism, unbounded history, a simple but flexible external structured file format called TIM (Tycho Information Models), and a simple serialization mechanism. Its subclasses implement application-specific models, such as storing user preferences, information about classes used by the documentation generator and class browser, indexes into the file system, and so on. There are also a set of graph classes which, although currently written in Itcl, we hope to eventually have written in Java.

Models are based on TIM (Tycho Information Models), which is a meta-data format that is intended to encourage clean representations of data, both in in-memory objects and in an external file representation. It is loosely based on the concepts of Object Modeling Technique (OMT) [14]: a model is a collection of entities and links between entities. Each entity and link has a unique name, value, and a list of attributes. Entities can be nested, so TIM is naturally hierarchical. A small TIM (for a dataflow graph) is:

```
vertex a {
    port out -tokencount 2 -type output
}
vertex b {
    port in-0
    port in-1
    port out -type output
}
edge a out b in-0 -initialdelay 0
```

Models contain a straight-forward implementation of the publish-and-subscribe pattern (also known as the *Observer* pattern [5]): any view that is interested in a model can subscribe to it, and will be notified of any updates to the model. The prototype graph editor in figure 3, for example, has two models: one for the graph, and one for the canvas layout. We have demonstrated multiple views editing the same two models. The concept is pervasive throughout Tycho – all major widgets, for example, are subscribed to the user preferences model. When a user changes, say, the font used in menu buttons, all menus are automatically updated with the new font.

Because user interfaces typically allow a user to undo and redo operations, Model implements a reasonably flexible, unbounded, undo and redo mechanism. Each method that changes a model is required to return a script that will undo the change. Both the method call and arguments and the undo script are stored in linear arrays.

4.2 The Displayer-View pattern

Early in its life, Tycho had two classes for every type of editor: one for the top-level window with menu and key bindings, and one containing the editing widget. We simplified this considerably by adopting a pattern we call Displayer-View, in which there is only one top-level window class (the Displayer). One or more views can place themselves into the Displayer and request access to a menu bar, tool bar, and status bar. The HTML window in figure 1 shows all three of these bars.

This approach gives us the flexibility to create widgets that can be placed into Displayers in new

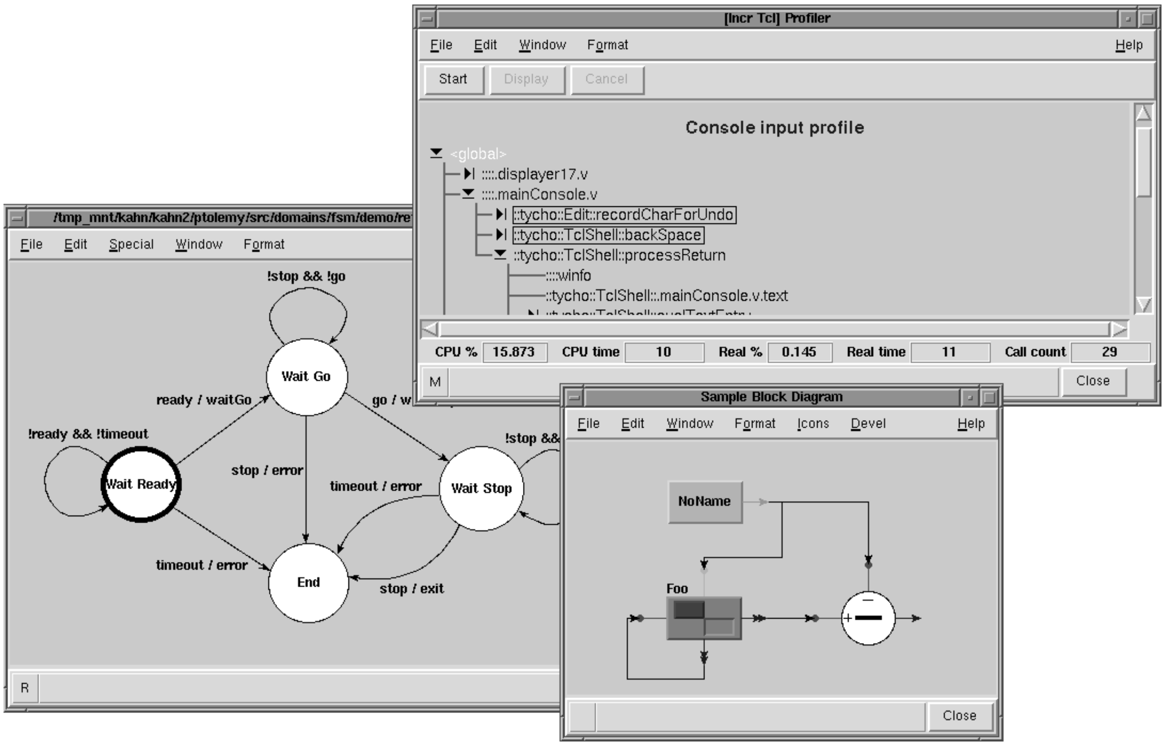

Figure 3: Some of Tycho's graphical editors

combinations. For example, a member of our group (Cliff Cordeiro) is working on a class browser that combines an HTML with a class index. We anticipate many more uses of this pattern in the future.

5 Graphical editing support

Although Tycho was originally intended only to serve as a user interface to Ptolemy. We now see Tycho as an opportunity to experiment with design visualization, broadening the perspective beyond a block-diagram perspective, and exploring new visual and mixed visual/textual syntaxes for design representation and understanding.

The infrastructure to support this vision falls into two categories. First is a set of classes that implement sophisticated canvas-like capabilities: hierarchical canvas items, abstractions for user interactions, and a graphical selection analogous to the textual selection of the Tk text widget. The second is a set of complete graphical editors. The editors are still early in development, but already include a simple finite-state-machine editor, a graphical class hierarchy display, and a graphical interface to the Tcl profiler from the TclX package [9] (see figures 1 and 3).

As much as possible, we have tried to capture functionality needed by typical graphical editors in Itcl classes. Tycho's enhanced canvas, which we call a *slate*, adds hierarchical items to Tk's standard canvas. The slate is implemented entirely in Itcl, and so should work with other canvas extensions such as the Dash patch [13]. The slate uses canvas tags to enable a collection of canvas items to be treated as a single complex item. All canvas commands functions correctly with single canvas items or complex slate items. Each type of complex item requires that a fairly small class be written to implement operations such as `create` and `coords`. We have a small library of such items, such as 3D rectangles and polygons, items with labels and scalable graphics on them (which we use for icons in graphical block diagrams), and self-routing lines (again for block diagrams).

Figure 4 illustrates a hierarchy of visual items. The root item contains a number of items within itself, one of which is an "icon" item that recursively includes two other complex items. It also contains a text label and a "terminal" item for connecting lines. The semantics of bindings within this visual hierarchy is not as simple as first appears: if the top-level item has tag *foo* and the terminal has tag *bar*, should the terminal also respond to events bound to *foo*? We have chosen to answer "no." So far,

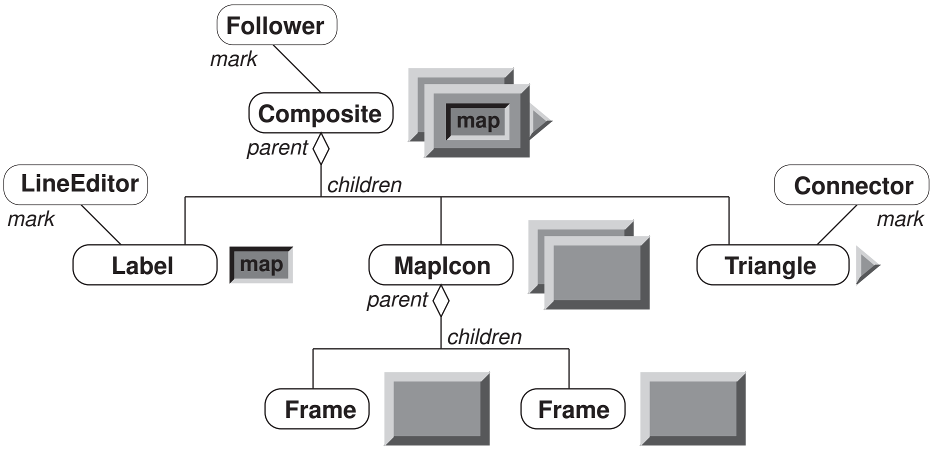

Figure 4: A hierarchical item marked with interactors

our experience indicates that this is the right decision both for implementation and use. In this example, the terminal items will therefore respond to events differently to the other items that make up the icon: if the mouse is dragged over a terminal, a new arrowed line is drawn to be connected to another terminal, whereas dragging the mouse on any other part of the item moves the whole icon, including the terminals and the ends of connected lines.

Figure 4 also shows user interaction objects, called *interactors* [12]. Each interactor captures a particular pattern of interaction. For example, a *Follower* class follows the mouse – by default, attaching an interactor to a canvas tag with a statement like `$follower bind icon` will make all items tagged "icon" draggable with the mouse. Of course, this is not difficult to do with Tk canvas bindings – but by encapsulating this into a class, we can inherit and compose interactors to produce more complex interactions. For example, subclasses of *Follower* keep items within a certain area of the screen, snap movement to a grid, and implement drag-and-drop mechanisms (the *Connector* in the figure). A *LineEditor* interactor edits text labels; a *Selector* interactor implements a graphical selection mechanism similar to that found in Macintosh drawing packages. Combined with the model-view architecture (section 4.1), we believe the visual hierarchy and interactors provide a powerful toolkit for building complex interactive graphical editors.

6 Tycho and Java

We have been exploring the integration of Tycho with Java. The combination of a high-level object-oriented scripting language, (Itcl) with an object-oriented system programming language (Java) is very attractive. Java in itself is attractive for its clean syntax and semantics, its support for distributed programming, and its high degree of portability. Members of the Ptolemy group are currently implementing an exploratory dataflow simulation engine in Java, which will be accessed from a Tycho graphical interface.

We based our Tycho-Java interface on version 0.4 of Sun's experimental Tcl-Java interface [15]. Not all features of Tcl work correctly in this implementation—`exec`, for example, is broken, and the exception handling does not work correctly with JDK1.1 – but it works well enough for us to explore the integration of Tcl and Java. We layered our own Java classes above the Tcl-Java interface to provide a simpler Tcl interface to Java programmers. We register a Tcl command called `::java::new`; this command creates a Java object of the requested class, a wrapper object that converts arguments into the appropriate types, and a Tcl command to access the object. For example, in Tcl, we could write:

```
::java::new tycho.TclExample foo
```

which create the Java objects and registers the Tcl command `foo`. In the TclExample class (in the Java package named `tycho`), a method might look like this:

```
public Double example1(Interp interp,
             Integer iobj, Double dobj ) {
    int i = iobj.intValue();
    double d = dobj.doubleValue();
    return new Double(i + d);
}
```

To call the method, we simply execute the Tcl code:

```
foo example1 5 3.2
```

which prints "8.2" as its result. The wrapper object uses Java's reflection API (`java.lang.reflect`) to find out the argument types of the method being called, converts the strings passed from Tcl into objects of the correct types, and then calls the correct method. Although there is some overhead in this approach, the interface is much cleaner than that provided by the Tcl-Java interface alone. With the Tcl8.0 byte-compiler, we think this style of interface will probably be needed to take advantage of Tcl native types.

As an alternative experiment, we used `tksteal` [3] to re-parent the Java `appletviewer` program into `itkwish`, but found that `appletviewer` didn't have the proper command-line interface to make this work smoothly.

In general, we would like to use [incr Tcl] (and [incr Tk]) for our user interface components, and Java for all new "back-end" development, including complex data structures and simulation tools. Although we have considered the possibility of using Java only, we would much prefer to integrate [incr Tcl] and Java. Apart from saving us a lot of rewriting (Java was still very new when Tycho was started), the Java user interface support is not as mature as Tk, and we rely heavily on Tk's more complex widgets, such as the text widget and the canvas widget.

In our view, it is unfortunate that Sun has not made a bigger effort to integrate Java and Tcl/Tk. Tk is unquestionably superior to Java's current UI support, and Java is the logical choice for future non-UI development. The Java UI is developing rapidly, however, and if we cannot successfully integrate Java and Tcl/Tk, then we may end up moving away from Tcl/Tk as the Java UI components mature. In our opinion, the most important efforts Sun can make to ensure the future success of Tcl and Tk are to (i) provide adequate support for object-oriented extensions to Tcl such as [incr Tcl], and (ii) to provide a seamless, efficient, and platform-independent interface to Java.

7 Implementation notes

7.1 Binary compatibility

One of the key factors in choosing implementation strategies for various features is that we want the Tycho core to run on any platform with no binary dependencies. To fulfill this requirement, we have made certain that Tycho will run within a standard `itkwish`. We have maintained a strong resistance to packages that require compilation to binaries, allowing this only if a) it is essential to gaining certain functionality and b) it is not going to significantly affect Tycho's cross-platform portability. The Tcl profiler from TclX [9], which we load into Tycho to work behind a graphical profile display, is a good example.

Another example of a C-coded package that adds useful (but not essential) functionality is support for integrating a Tycho-based custom user interface to a C program generated by Ptolemy. One of Ptolemy's features is an ability to generate C code from signal processing block diagrams. We have added a Tycho "target" to Ptolemy, which allows it to generate Tcl packages. Each package implements a simple execution protocol that allows us to load multiple packages into Tycho and interleave execution (while still keeping the user interface live). The screen shot in figure 1 show a real-time digital audio application running within Tycho – this ten-band stereo graphic equalizer easily runs at 44.1 kHz (CD quality) on a Sun UltraSparc.

A third example where we require binary support is to run Tycho with Java (section 6). Java is becoming more and more important in Tycho, however, and the need for a customized binary interface compromises our portability goals (despite the portability of the two languages separately). It is vitally important that Sun include a Java interface in the Tcl core.

7.2 Choice of widgets

Tycho uses its own library of mega-widgets. Tycho's widget library includes list and file browsers, font and color selection, alert boxes, an HTML displayer message window (for help messages), and so on. With most of these classes, we aim not to provide a complete "use-as-is" widget, but a foundation for customizing by inheritance and composition. For example, one of the key classes in the dialog box hierarchy is called *Query*: it contains a button box and an uncommitted frame for labeled entry widgets. A suite of methods add entry fields, check-buttons, option menus, and so on, to this frame. Many of the dialog classes – including the search dialog shown in figure 1 – inherit from this class and configure these fields in the constructor. This kind of extensibility is one of the great strengths of the object-oriented approach.

Throughout Tycho's development, we have faced the decision on whether to use existing widgets or to develop our own. The most suitable library is [incr widgets], or Iwidgets [16], a library of widgets written in [incr Tcl] that includes labeled widgets, scrolled canvas and text widgets, dialog boxes, a tabbed notebook, button and radio boxes, and so on. Other libraries, such as Tix and BLT, clashed with our "no-binaries" policy.

Tycho does not currently use any of the Iwidgets classes. We are not at all opposed to Iwidgets, and believe them to be a very useful set of widgets. In particular, we have used many ideas from Iwidgets and freely acknowledge our debt to the Iwidgets authors. In the future, widgets like the tabbed notebook may well be used within Tycho where we feel it is appropriate. When we began work on Tycho, however, Iwidgets was also early in development, and we found we had problems with many of them that were most easily fixed by writing our own. For example, moving the focus ring in the Iwidgets button box made the whole window adjust itself – visually, a very disconcerting effect. The Tycho button box widget does not do this.

Part of the reason that the Tycho widgets avoid some of the problems the Iwidgets authors faced is that they are somewhat more dedicated to our needs within the Tycho environment. We have not tried to deal with all possible uses, nor to provide all possible widgets. Instead, we have been able to choose what we felt was an adequate design where it was needed. Tycho widgets generally have far fewer configuration options than Iwidgets and more specific functionality. Iwidgets, for example, provides a scrolled canvas – the equivalent functionality in Tycho is a complete view with support for menus, printing, graphical selection, and so on.

There are secondary considerations that have worked against adopting Iwidgets. Tycho widgets benefit from our automatic documentation system, which includes executable Tcl examples embedded in HTML windows. Tycho widgets also appear on the Tycho class diagram, and are thus more visible to a Tycho developer.

7.3 Cross-platform porting

Our current release (May 97) is Tycho0.1.1. We are in the beta phase of Tycho0.2, which will be final in June 1997.

On the Macintosh, Tycho0.2 requires Itcl2.2p2. Under Itcl2.2 on the Macintosh, Tycho0.1.1 was crashing during startup. Unfortunately, inserting `puts` statements would move the location of the crash around. Jim Ingham, (formerly at Lucent, now at Sun) determined that the problem was that the Itcl2.2 Macintosh binaries had too small a stack to load all of Tycho. Under Itcl2.2p2, the Macintosh port starts up on some machines, but we were never able to complete startup on the infamous PowerBook 5300c.

We've tested Tycho0.1.1 and Tycho0.2 under NT, and most features work. The biggest problem is that non-blocking I/O is not available in the prebuilt Windows binaries. Under Windows, certain features of Tycho that start up subprocesses using non-blocking I/O do not work as they do under Unix. Unfortunately, the Macintosh does not even have `exec`, so the Macintosh port is even more limited.

A major issue facing the Macintosh and NT ports is the presence of spaces in file names. Passing filename arguments that contain spaces can be a little tricky. Support for file names with spaces could be improved – for example, the Tk4.2 demos that come with the prebuilt Itcl2.2 binaries do not work if there is a space in the pathname. Configuring Tcl8.0b under Unix also fails if there is a space in the prefix option to configure.

8 Concluding remarks

For user interface development, Tk is flexible and powerful. We have found that [incr Tcl] and [incr Tk] greatly improve the structure and clarity of medium-sized user-interface programs (Tycho is currently 64,000 lines of [incr Tcl]). Compared to the legacy Tcl/Tk code in Ptolemy, the Tycho code is marvelously well-organized. We would strongly recommend [incr Tcl] and [incr Tk] for Tk user-interface development for any substantial program.

[incr Tcl] and [incr Tk] are not without shortcomings. Many are largely inherited from Tcl/Tk: poor performance (which the new byte-compiler will greatly alleviate), and extremely lax parsing. In programs of this scale, run-time errors caused by an inadvertent extra parenthesis (for example), are extremely annoying and do nothing to give us confidence in the stability and robustness of our code. We have had problems with Tk's focus mechanism and the interaction of [incr Tcl] objects with Tk's update mechanism. As much as possible, we have worked around these, and other developers may wish to use the work-arounds incorporated into Tycho.

On the whole though, our experience in this project has been very positive and we are excited about future prospects. We urge Sun's Tcl/Tk group to focus on integrating Tcl/Tk (and [incr Tcl]/[incr Tk]) and Java into a seamless scripting/user-interface/code-development environment. This we see as the logical future of platform-independent computing.

Tycho is freely available under the unrestrictive UC Berkeley license – see the Tycho home page: `http://ptolemy.eecs.berkeley.edu/tycho/`.

Acknowledgments

Tycho is the work of many people. Contributors to the Tycho project include Kevin Chang, Wan-Teh Chang, Cliff Cordeiro, Wei-Jen Huang, Joel King, Farhana Sheikh, and Mario Jorge Silva. Key infrastructure without which this project would not have been possible has been developed by: Michael McLennan of Bell Labs ([incr Tcl]/[incr Tk]), John Ousterhout of U.C. Berkeley and Sun Microsystems (Tcl/Tk) Stephen Uhler of Sun Microsystems (HTML library), and Mark L. Ulferts of DSC Communications Corp ([incr Widgets]).

Tycho is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Dolby Laboratories, Hewlett Packard, Hitachi, Hughes Space and Communications, LG Electronics, Lockheed Martin ATL, NEC, Philips, and Rockwell.

References

[1] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4, April 1994. Special issue on Simulation Software Development.

[2] Dave Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, 1995.

[3] Sven Delmas and Juergen Nickelsen. Information on TkSteal. `http://www.cimetrix.com/sven/tksteal.html`.

[4] Lisa Friendly. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design '95*. Sun Microsystems, Inc, 1995. `http://www.javasoft.com/doc/api_documentation.html#javadoc`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reuse in Object Oriented Software*. Addison-Wesley, 1994.

[6] The Ptolemy Group. The Ptolemy home page. `http://ptolemy.eecs.berkeley.edu/`.

[7] The Ptolemy Group. The Tycho home page. `http://ptolemy.eecs.berkeley.edu/tycho/`.

[8] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. `http://ptolemy.eecs.berkeley.edu/papers/overview/`, March 1994.

[9] Karl Lehenbauer and Mark Diekhans. The TclX distribution. `http://www.neosoft.com/tcl/ftparchive/TclX/`.

[10] Udi Manber, Sun Wu, and Burra Gopal. Glimpse: A tool to search entire file systems. `http://glimpse.cs.arizona.edu/`.

[11] Michael J. McLennan. The [incr Tcl] home page. `http://www.tcltk.com/itcl/`.

[12] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[13] Jan Nijtmans. Dashed and stippled outlines in Tk8.0a2 (Tk4.2p2, Itcl2.2). `http://www.cogsci.kun.nl/~nijtmans/tcl/patch.html`.

[14] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[15] Scott Stanton and Ken Corey. The TclJava demonstration. `ftp://ftp.sunlabs.com/pub/tcl/tcljava0.4.tar.gz`.

[16] Sue Yockey, Mark Ulferts, Bret Schuhmacher, John Sigler, and Alfredo Jahn. [incr Widgets]. `http://www.tcltk.com/iwidgets/index.html/`.