# Extending Traces with OAT[1]: an Object Attribute Trace package for Tcl/Tk[2]

Alex Safonov, Joseph A. Konstan, John V. Carlis and Brian Bailey
*Department of Computer Science*
*University of Minnesota*
*{safonov,konstan,carlis,bailey}@cs.umn.edu*

## Abstract

Tcl supports variable traces, which associate arbitrary scripts with variable reads, writes and unsets. We developed OAT (Object Attribute Traces), a protocol for extending traces to attributes of arbitrary Tcl "objects." We wrote several OAT-based extensions including TkOAT, which provides traces on attributes of Tk widgets and canvas items. The OAT protocol and derived extensions bring the benefits of more expressive constraints to Tcl/Tk applications by providing extended traces. OAT requires no changes to the Tcl core and is implemented as a loadable library; OAT-based extended trace packages introduce minimal changes to the code of existing extensions (Tk, CMT, etc.). The new version of our formula manager, TclProp, takes advantage of extended traces provided by OAT.

## 1. Introduction

Tcl's trace mechanism allows the Tcl script programmer to specify arbitrary scripts to be executed when a given variable is read, written, or unset. It provides, among other things, a good means for propagating changes to variables [Sah95]. For instance, Tcl uses traces internally for the following:

- tracking changes to the `tcl_precision` variable, to update the floating-point precision and printing format in the interpreter.
- monitoring the `env` array to propagate changes to the corresponding environment variables.
- supporting linked Tcl and C variables.
- implementing the `vwait` command.

The built-in trace mechanism is limited to variables only. This limitation becomes significant, for instance, when one attempts to use traces and formulas with UI widgets. It is often desirable to detect changes to the state of Tk widgets, or to link widget attributes with formulas. Because many attributes of Tk widgets are not associated with variables, they are not traceable. Examples of such widget attributes include:

- button state, normal or disabled
- state of a menu item, normal or disabled
- button or label color and bitmap
- the number of items in a listbox

We propose OAT, a generic protocol for extending Tcl traces. We target two types of users: Tcl script programmers and Tcl extension developers. Script programmers will benefit from the OAT-based extended traces, because the trace-based Tcl code is more compact and easier to maintain. Extension developers can use OAT to make their objects traceable, to bring the benefits of declarative trace-based programming to users of their extensions. We discuss our experience of making Tk widgets and CMT clocks traceable. Finally, we describe the OAT implementation.

## 2. Extended traces

The Listbox Pager shown in Figure 1 demonstrates the usefulness of traces set directly on widget attributes. The "Prev Page" and "Next Page" buttons scroll the contents of the listbox by the number of visible lines. They are disabled when the listbox is positioned on the last page (starting at item 5) and the first page (starting at item 0), respectively. The state of the pager buttons depends on the following three attributes of the listbox:

1. the total number of items
2. the number of visible items
3. the number of the first visible item

[1] One of the reviewers suggested that we come up with a different name for the OAT package, so we renamed our package FATCAT: Flexible Architecture for Tcl Constraints And Traces.

[2] Partially supported by NSF grants EBN-9419233 and IRI-9410470, and a grant from the Minnesota Distributed Multimedia Research Center.

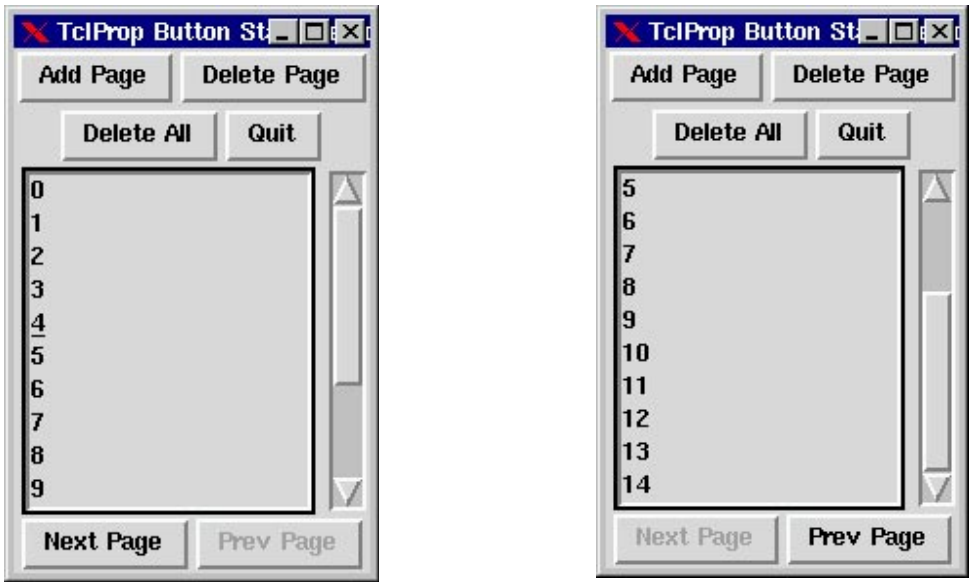


Figure 1. A listbox pager using extended traces: first and last pages

Updates to these three listbox attributes can be scattered throughout the script code and widget callbacks. Without extended traces, these updates must be followed by the code that enables and disables the “Prev Page” and “Next Page” buttons. With traces on the listbox attributes available, the Tcl programmer only needs to specify the code to set the state of the pager button once, in the trace callback. The working code in Figure 2 creates the extended traces on listbox attributes, and defines a procedure to check whether the listbox is on the last page, and to enable or disable the buttons accordingly. The trace callback that controls the state of the “Prev Page” button is similar. Since listbox attributes are not traceable in stock Tk and no variables are associated with them, this example demonstrates the need for extended traces for UI development.

Extended traces are useful for canvas item "geometry management". Suppose we would like to keep a canvas rectangle twice the size of another one as the latter is resized. We create the trace on the `coords` attribute of the source rectangle, and associate with it the code to resize the target rectangle. Similarly, the trace on canvas item `coords` can maintain its width-to-height ratio constant, ensuring, for instance, that a rectangle remains square.

Extended traces on canvas attributes also help to maintain geometric relationships among them. For instance, graph drawing packages [Ellson96]

```
# create extended traces on topIndex, numElements, and fullLines
# attributes of listbox widget .lb
trace widget .lb topIndex    w setNextPageState
trace widget .lb numElements w setNextPageState
trace widget .lb fullLines   w setNextPageState

# define trace callback
proc setNextPageState {nameSpace widgName attrName op} {

    set fraction2 [lindex [$widgName yview] 1]
    if {$fraction2 < 1} {
        .nextPage conf -state normal
    } else {
        .nextPage conf -state disabled
    }
}
```

Figure 2. Code for trace-based “Next Page” button

and programs for structured drawing benefit from the ability to keep canvas items attached as they are dragged. A line is attached to a circle by creating traces on both line and circle coordinates[3]. The code associated with the traces moves one item as the other is dragged.

## 3. Traceable types and the OAT protocol

These examples and our experience with Tcl/Tk development motivated us to create a protocol for defining traces on attributes of arbitrary Tcl "objects" [Roseman95]. We call an object type enhanced to support traces on its attributes a *traceable type*. Variables constitute a traceable type with only one (implicit) attribute: their value. Other traceable types we created are Tk widgets, Tk canvas items, CMT clocks, and [incr Tcl] namespaced variables and objects (work on the latter is still in progress).

Our primary goals in the development of OAT were:

1. to define a clean protocol for easily adding traces to new and existing extensions that employ the concept of state-holding objects;
2. to extend the benefits of traces to arbitrary Tcl objects.
3. to make OAT and OAT-based extensions easy to integrate with existing Tcl/Tk installations.

As a proof of concept and for general use, we developed TkOAT, supporting traces on Tk widgets and canvas attributes, and MediOAT, adding traces to CMT clocks. We tried to conform as much as possible to the Tcl interface provided by the trace command, and internal C interfaces, since we found these well-designed and implemented. To ease integration of OAT and OAT-based extensions with Tcl/Tk installations, we created the extensions as loadable libraries, and made changes to the original code only where it was absolutely necessary.

The OAT protocol consists of two parts:

- Tcl API, used by the Tcl script programmer. It defines creation, querying, and deletion of extended traces on objects of traceable types from Tcl code.
- C API, used by the Tcl extension developer. The OAT C API defines a protocol for creating new traceable types, and checking and triggering traces on attributes of traceable type objects.

[3] While the coordinates are not, strictly speaking, configurable "attributes" of canvas items, for cases like these it is desirable to treat them as such.

TclProp [Iyengar95], a trace-based formula manager for Tcl, was updated to take advantage of extended traces. The new TclProp API includes type identification for objects whose attributes are terms in a formula.

### 3.1 OAT Tcl API

The OAT Tcl API permits a programmer to manipulate extended traces from Tcl scripts. The interface is modeled on variable traces. As with variables, each traceable type has three subcommands for the trace command: to create, query, and delete a trace. Code in Figure 2 shows an example of creating traces on attributes of listbox `.lb`; the "`widget`" keyword is the trace creation subcommand for the Tk widget traceable type. The general syntax of the extended trace command, shown in Figure 3, is the same as in stock Tcl. However, the "option" subcommand can be a trace keyword for any of the registered traceable types. The keywords for the subcommands are defined when a traceable type is created, as discussed below.

```
trace option ?arg arg ...?
```

Figure 3: The extended trace command syntax

A Tcl programmer can query OAT for all registered traceable types; this is accomplished with the "`oat`" command. It returns the names of all registered traceable types; the list always includes the built-in traceable type, `variable`. Suppose the TkOAT extension is loaded, providing traces on widget and canvas attributes. The `oat` command will return the following list: `"variable widget citem."` Here "`widget`" and "`citem`" are additional traceable types registered by TkOAT. The "`oat typename`" command returns the subcommand keywords for a registered traceable type, where "`typename`" is a name of a traceable type. The `"oat widget"` command will return the trace subcommands for traceable type widget: the list `"widget winfo wdelete."`

### 3.2 OAT C API

The traceable type creation protocol permits a writer or a maintainer of a Tcl extension to define new traceable types. The creation of a traceable type involves three main steps:

- **extend the trace command**: the trace command is extended to accept keywords for new sub-

commands that create, query, and delete traces on this type.

- **register the callback function**: the OAT-supplied or custom C callback is registered to be called when these keywords are supplied with the trace command.
- **insert checks for traces**: checks for traces are inserted in the attribute update code for the traceable type.

**Extend the trace command**. A traceable type can be defined by the extension developer at different levels of detail. At a minimum, the name of the new traceable type must be supplied. The OAT library will automatically generate the subcommand keywords for the trace command from the type name. For instance, if `"gadget"` is specified as a traceable type name, the keywords to create, query, and delete traces on it will be `"gadget"`, `"gadgetinfo"`, and `"gadgetdelete"`, respectively. If desired, these three keywords can be explicitly specified. As an example, Tk widgets form a traceable type with trace subcommands `"widget,"` `"winfo,"` and `"wdelete."`. Figure 4 shows the OAT traceable type data structure, along with field names, types, and brief content descriptions.

**Register the callback function**. The developer can use the standard OAT-supplied callback for trace subcommands, or write a custom one. The first approach significantly simplifies creating a traceable type. The extension developer should use the OAT-supplied callback if all extension objects are "global" like Tk widgets, that is, are not contained within a namespace.

Object systems can require additional information to identify an object being traced. For example, traces on canvas item attributes require both a canvas name and an item tag or id to uniquely identify the traced attribute. To support this behavior, OAT allows the extension developer to specify a custom C function that is called when the Tcl interpreter processes the trace subcommands for this type. The prototype for the OAT custom trace callback is shown on Figure 5. We are investigating how to make OAT-supplied trace callbacks more general so that they accommodate objects with namespaces.

**Insert checks for traces**. The extension developer needs to insert the check for traces in the C code where extension object attributes are updated. This check is done in a single line of code, as shown on Figure6.

| Field Name | Field Type | Comment |
|---|---|---|
| `typeName` | `char*` | traceable type name |
| `traceCreate` | `char*` | keyword for trace command |
| `traceInfo` | `char*` | keyword for trace command |
| `traceDelete` | `char*` | keyword for trace command |
| `traceCmdProc` | `function pointer` | function to call when trace subcommand for this type is supplied, or `NULL` (use default callback function) |

Figure 4: A traceable type data structure

```
typedef int (Oat_CmdProc) (ClientData dummy,
                           Tcl_Interp* interp,
                           int argc, char* argv[]);
```

Figure 5. Type definition for the custom OAT trace callback.

```
/* Call traces for widget or canvas item attribute being configured. */
Oat_CallObjTraces(interp, objRec, specPtr->argvName, TCL_TRACE_WRITES);
```

Figure 6. Adding check for extended traces to `tkConfig.c`

```
char* Oat_CallObjTraces (
      Tcl_Interp* interp,   /* current interpreter              */
      char*       objPtr,   /* pointer to object data structure */
      char*       attrName, /* attribute name                   */
      int         flags);   /* currently TCL_TRACE_WRITES only  */
```

Figure 7. Prototype for the OAT function that checks and triggers traces

For Tk widget and canvas item traces, which share the code to update attributes, the call to `Oat_CallObjTraces()` is inserted at the end of `DoConfig()`. This is the only place where the core code needs to be modified to support widget attribute traces in Tk. Figure 7 shows the prototype for function `Oat_CallObjTraces()` that specifies what information uniquely identifies the object and the attribute.

When the trace command is used in a Tcl script, the OAT code searches for the supplied subcommand keyword in the table of registered traceable types. When a matching keyword is found, a standard or custom function to manipulate a trace is executed. If the keyword does not match any of the registered traceable types, the trace command returns an error. This implementation supports traces on variables in a uniform interface, since variables are defined as a traceable type with subcommands `"variable,"` `"vinfo,"` and `"vdelete,"` and with the Tcl-supplied C callback, `Tcl_TraceCmd()`, that processes these subcommands.

## 4. MediOAT: a traceable type extension for Continuous Media Toolkit

CMT [Rowe92], a multimedia toolkit from Berkeley, has an object system similar to Tk widgets. Examples of CMT objects are clocks, packet sources and destinations, and media play objects. Unlike Tk, however, the CMT object implementation does not use a single `ConfigureObject()` function; instead, each type of CMT objects parses its command options. For the OAT implementation of extended traces, this means that checks for traces need to be inserted in the attribute manipulation function for each type of CMT object. Based on our experiences with CMT multimedia applications, we decided that traces on clock attributes - speed and value - would be most useful. Traces on clock attributes trigger code when the time in a multimedia presentation "jumps", stops, or starts to move at a different rate. Our current version of MediOAT supports traces on attributes of CMT clocks. We wrote VCR control and jog shuttle control megawidgets based on MediOAT. Figure 8 shows GIF animation with VCR and jog shuttle controls. The control buttons - stop, play, fast forward, reverse - set the `"-speed"` attribute of the shared CMT clock. The trace callback, associated with clock speed updates, highlights the VCR buttons appropriately. The resulting code is compact; without traces, callback for each control button would have to know what other buttons to highlight.

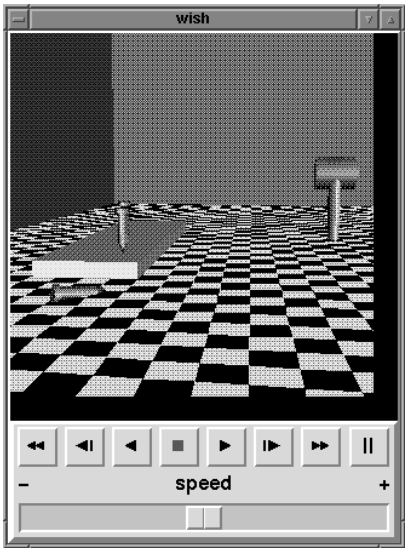


Figure 8. A MediOAT-based VCR controls (shown stopped: stop button disabled)

```
trace lts $lts -speed w \
   [list VCR_LTSChanged $id]
```

Figure 9. Creating a trace on a CMT clock; procedure `VCR_LTSChanged` manipulates the VCR buttons state.

## 5. OAT Implementation

Variable traces rely on special fields in the C structure describing a Tcl variable. This approach was not possible in OAT, since object systems currently do not reserve space for storing trace information. In Tk, there is not even a single data structure common to all widgets. We decided to use a hashtable associated

with the interpreter to hold trace information. The hashtable string key consists of two components: the memory pointer of the C data structure describing the traced object, and the name of the attribute. To generate a string key, the binary pointer value is converted into its hexadecimal string representation. Why did we use the object pointer instead of name? Object name is not always available when its attributes are updated. To use the pointer for a part of the hashtable key, we assume that each traceable object is represented as a data structure with a unique address. We believe this assumption will hold as Tcl is upgraded with new versions. When the dual-ported object system was introduced in Tcl 8.0, we updated the OAT code to use the pointer to the object, rather than the string, representation in Tcl version 8 and above.

We did not observe performance degradation on trace manipulations from the OAT protocol. To evaluate the performance impact of OAT, we timed creation and querying of variable traces in the following two environments:

- original: Tcl7.6p2 with Jan Nijtmans's plus-patch [Nijtmans97], and Tk4.2p2 dynamically loaded
- OAT- and TkOAT-enhanced: Tcl7.6p2 with Jan Nijtmans's plus-patch, OAT, Tk4.2p2, and TkOAT loaded.

Timings were generated on several UNIX platforms: Solaris, Linux, and Irix. Both the original Tcl interpreter, and the interpreter with OAT and TkOAT loaded, produced essentially the same timings. We conclude that the lookup of additional trace keywords in the OAT library code does not lead to any noticeable performance degradation.

As we used OAT and TkOAT, we found it useful to trace components of object state that are not configurable attributes. The Listbox Pager example in Figure 1 relies on traces on such "virtual" listbox attributes: the number of listbox items, the number of visible items, and the first visible item. We concluded that supporting traces on these attributes was valuable enough to make further modifications to Tk: insert checks for traces in the listbox code. However, this violated our goal of non-invasive Tk extension. We hope that as traces and constraints are used more widely in Tcl/Tk applications, more attributes of widgets and objects will be "exported" by developers. The Tcl programmer will be able to use traces on a wider class of objects and attributes.

OAT was inspired by the variable traces in Tcl. TkOAT was made feasible largely by a single location in the Tk code where configurable widget and canvas item attributes are set. While MediOAT demonstrates that it is possible to insert OAT hooks in any place where object attributes are potentially updated, this defeats our goal of minimally invasive code changes. We encourage all extension developers who write object-like systems to adopt the Tk model, where names and types of object attributes are defined in a C data structure, and all attribute queries and updates go through function similar to Tk's `ConfigureWidget()`.

## 6. OAT and TclProp

While traces present a powerful abstraction for UI programming, they can be inconveniently low-level to be expressive. TclProp, a trace-based, script-only formula manager and data propagation engine for Tcl is described in [Iyengar95]. One of our goals in developing OAT was extending the benefits of TclProp formulas to arbitrary Tcl objects. TclProp was rewritten to be accommodate formulas on new types of objects. One main change to the TclProp API was the addition of traceable type names in formulas - these are needed to invoke the appropriate trace subcommands inside TclProp. TclProp uses the `"oat typeName"` command to retrieve the trace keywords, as described above. To support extended traces, TclProp also needs to know how to read and write attributes of traceable types. We use a Tcl array to associate traceable types with scripts that access attributes of these types. For example, the read code for the traceable type, Tk widget, is shown on Figure 10.

```
#the slot in the TP_traceableType array associates the traceable type
#name, widget, with the name of the procedure taking the names of a Tk
#widget and its attribute, and returning the code to read the value of
#this attribute.

set TP_traceableType(widget,read)  "widgReadFunc"

proc widgReadFunc {widgName attrName} {
      return "$widgName cget $attrName"
}
```

Figure 10. Extending TclProp to new traceable types: a Tk widget-specific attribute read procedure

```
TP_formula \
   "citem .canv rect1 coords" \                  # formula destination
      [list \                                    # list of formula sources
         "rC [list citem .canv rect coords]"] \       # source 1
   {list [expr [lindex $rC 0]+100] [lindex $rC 1] \   # formula code
         [expr [lindex $rC 2]+100] [lindex $rC 3]}
```

Figure 11. TclProp formula with Tk canvas item attributes as formula destination and source. Comments are for explanation only.

The Tcl interpreter evaluates procedure "`widgReadFunc`" when a TclProp formula is created, placing the code to read the value of the widget attribute into the formula code. Since the procedure is evaluated at formula creation time and not at formula propagation time, no additional performance penalty is incurred by the extension of TclProp to new traceable types.

Code for a TclProp formula typically accesses the values of variables and object attributes that the formula depends on. While variables reads in Tcl are compact, code to read the values of object attributes can be quite verbose. For example, to compute the width of a canvas rectangle, the following expression is needed:

```
{expr [lindex [.canv rect coords] 2] - \
      [lindex [.canv rect coords] 0]}
```

We enhanced TclProp formula syntax with *tags* to support a more compact access to object attributes in the formula code. Tags are associated with object attributes on which the formula depends, and are replaced with the actual attribute read code in the formula body. In the example above, if the tag "`rC`" is specified for the "`.canv rect coords`" rectangle attribute in the formula, the code above is simplified as follows:

```
 {expr [lindex $rC 2] - [lindex $rC 0]}
```

The complete code for the TclProp formula ensuring that canvas rectangle `rect1` is offset by 100 pixels in x relative to `rect`, is shown on Figure 11.

In our experience with OAT and TclProp, we found the separation of trace detection and formula propagation very useful. A propagation model different from eager propagation model in TclProp can be built on top of Tcl and OAT traces. We experimented with several models, and implemented the lazy propagation alternative to TclProp, TclLazy, on top of OAT in a few hours. In the lazy propagation, variables in formulas are marked invalid on writes. A formula is evaluated only when an up-to-date value of its left-hand side is needed. The lazy model is appropriate when writes significantly dominate reads. We believe that the separation of traces and constraint propagation can be adopted in other scripting languages, such as Perl, to provide easier development of constraint engines.

## 7. Future Work

We use constraint programming and traces in our Tcl/Tk development extensively [Safonov96], and would like to see OAT and OAT-based extensions more widely used. We plan to work in the following four areas:

1. **Make more types of extension objects traceable**. We are currently considering adding traces to CMT media segment objects; this will facili-

tate the development of CMT-based timeline editors. We also plan to make objects traceable in VTk, a Tk-based 3D graphics extension, and GroupKit [Roseman96], a groupware for Tcl/Tk.

2. **Make adding traceability to objects easier**. Currently, the extension developer needs to modify code to insert checks for traces. While these changes are minimal, recompilation is still necessary. We plan to develop a protocol that will allow extension developers to insert hooks into the code that are later bound to the OAT function `Oat_CallObjTraces()`.
3. **Add object-oriented features to OAT**, based on [incr Tcl]. We see benefits in initializing class and object attributes to formulas rather than values, and in inheriting traces and constraints.
4. **Investigate other languages and environments for adding the trace protocol.** We have considered Perl as the potential target for adding the trace mechanism and trace-based constraint manager.

## References


[Ellson96] John Ellson and Stephen North. TclDG - a Tcl Extension for Dynamic Graphs. In *Proceedings of the 4th Tcl/Tk Workshop*, p. 37. USENIX Assoc; Berkeley, CA, 1996.

[Iyengar95] Sunanda Iyengar and Joseph A. Konstan. TclProp: a data-propagation formula manager for Tcl and Tk. In *Proceedings of the 3rd Tcl/Tk Workshop*, p. 288. USENIX Assoc; Berkeley, CA, 1995.

[Nijtmans97] Jan Nijtmans. The Plus-patches. In *http://www.cogsci.kun.nl/tkpvm/pluspatch.html*

[Roseman95] Mark Roseman. When is an object not an object? In *Proceedings of the 3rd Tcl/Tk Workshop*, p. 197. USENIX Assoc; Berkeley, CA, 1995.

[Roseman96] Mark Roseman and Saul Greenberg. Building Real Time Groupware with GroupKit, a Groupware Toolkit. *ACM TOCHI*, March 1996.

[Rowe92] Lawrence A. Rowe and Brian C. Smith. A Continuous Media Player. In *Network and Operating System Support for Digital Audio and Video, Proceedings of the Third International Workshop on*, p. 376-86, 1992.

[Safonov96] Alex Safonov, Douglas Perrin, John Carlis, Joseph Konstan, John Riedl, and Robert Elde. Lessons from the Neighborhood Viewer: A Tool for Collaborative 3D Exploration of 2D Images. In *Proceedings of the 4th Tcl/Tk Workshop*, p. 203. USENIX Assoc; Berkeley, CA, 1996.

[Sah95] Adam Sah. Multiple Trace Composition and Its Uses. In *Proceedings of the 3rd Tcl/Tk Workshop*, page 288. USENIX Assoc; Berkeley, CA, 1995.