Assertions for the Tcl Language

Jonathan E. Cook
Department of Computer Science
New Mexico State University
Las Cruces, NM

Assertions for the Tcl Language

Jonathan E. Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
jcook@cs.nmsu.edu
http://www.cs.nmsu.edu/~jcook

Abstract

Assertions, even as simple as the C `assert` *macro, offer important self-checking properties to programs, and improve the robustness of software when they are used. This paper describes* AsserTcl, *an assertion package for the Tcl programming language. Our assertions take the form of commands in the program text, and cover point assertions about the computation state, assertions about procedure input values and the return value, and assertions about the values that variables may take on over their whole lifetime. In addition, universal and existential quantifiers are provided for both lists and arrays, not only for individual elements, but for sequences of elements as well.*

1 Introduction

Assertions—declarative annotations to a program that describe some property about the program and its state—are useful tools in producing robust software. Most programming languages are not designed to include assertions, with the notable exception being Eiffel [3], but efforts have produced annotation languages for Ada [2], C [5], Awk [1], and others. This paper describes assertions for the Tcl programming language [4].

Tcl is a popular, interpreted, "scripting" language. It is now being used to build large (tens and hundreds of KLOCs) systems, many of which are being used in the commercial sector. Unfortunately, few tools exist that help one build robust Tcl applications.

This is where assertions fit in. Assertions help developers write robust code by offering dynamic checking of program properties, and enhancing the program's testability. Our AsserTcl package provides assertions for the Tcl programming language. The assertions take the form of commands in the program text, and cover point assertions about the computation state, assertions about procedure input values and the return value, and assertions about the values that variables may take on over their whole lifetime, singly or in relation to other variables.

In addition, many desirable assertions in Tcl will be over an aggregate data structure, which in Tcl is a list or array. For this, we provide universal and existential quantifiers for both lists and arrays, not only for individual elements but for sequences of elements as well.

Section 2 describes our assertion commands and their meaning and use, and Section 3 describes the quantifiers and their use. Section 4 describes the interface for controlling the evaluation of assertions. Section 5 presents some pragmatic issues in adding assertions to the Tcl language, and describes the methods we used to implement assertions. Section 6 presents some examples in using the assertions and quantifiers. Section 7 evaluates the performance of a Tcl program that uses assertions. Finally, Section 8 concludes with some observations about assertions, the Tcl language and its future, and possible enhancements to the language to make it easier to develop debugging and analysis tools.

2 Assertion Commands

Our assertions for Tcl are commands that use a normal Tcl expression as an assertion about some portion of the program. The four assertion commands that we have added to Tcl are:

- *assert* is an assertion about the current state of computation. It is evaluated at the point it occurs in the source;
- *assume* is an assertion about the input values to a procedure. It is evaluated upon entry to a procedure;
- *assure* is an assertion about the output and return values of a procedure. It is evaluated upon exit from a procedure;
- *always* is an assertion about part of the state space (i.e., variables) of the program. It must always hold, and is evaluated each time one of the variables it depends on changes value.

These commands are summarized in Table 1.

The form for each of these assertions is:

General Form of Assertions	
assert-cmd expr ?fail-action? ?-default_also?	Each assertion evaluates *expr* at some point or points in the execution of the program. If the expression evaluates to true (nonzero), no action is taken. If false (0), *fail-action* is taken if it is specified, otherwise a general exception occurs. The *fail-action* can use break, continue, or return. Specifying the *-default_also* flag will force the general exception to occur after doing the *fail-action*.
Specific Assertion Commands	
assert *expr ?fail-action? ?-default_also?*	Point assertion: evaluates *expr* at the point of its specification, each time the program reaches that point.
assume *expr ?fail-action? ?-default_also?*	Procedure entry assertion: evaluates *expr* at the beginning of a procedure, and should be an assertion about the input parameters of the procedure (and global variables used). This command should be placed at the top of the procedure body, just after any *global* statements, so that it can be seen as part of the procedure specification.
assure *expr ?fail-action? ?-default_also?*	Procedure exit assertion: evaluates *expr* just before returning from a procedure, and should be an assertion about the return value of a procedure, and any side effects (global variables modified) of the procedure. Note that this command does not need to be placed at the end of the procedure, and should be placed at the top, after any *assume*s, so it can be seen as part of the procedure specification. The values of the input parameters at the time of procedure invocation are available through **_in* variables, one for each parameter. The return value of the procedure is available through the *return_val* variable.
always *varlist expr ?fail-action? ?-default_also?*	Program state assertion: evaluates *expr* every time one of the variables in *varlist* changes. The variables in *varlist* should be (a subset of) the variables in *expr*, and the expression should be a statement about the relationships that must hold or values those variables can take on. The assertion exists for the lifetime of the variables in *varlist*, and the *always* assertion can be used in procedure bodies for local variables, as well as for global variables.

Table 1: The four basic assertion commands for Tcl.

assert-cmd expression [failure-action] [-default_also]

where *assert-cmd* is one of *assert*, *assume*, or *assure*. The *always* assertion is:

always varlist expression [failure-action] [-default_also]

In both forms, *expression* is the Tcl expression to be evaluated. For the assertion to be satisfied, this expression must evaluate to true (nonzero). For the *always* assertion, *varlist* is a list of variables to activate the assertion on, such that when any one of them changes, the expression is evaluated. It does not have to include all the variables in the *always*,[1] but should not include any variables not in the expression.

Failure-action is an optional Tcl statement (block) that, if supplied, will be executed if the assertion fails. If no *failure-action* is specified, the default action of producing a general exception occurs, printing out the nature of the failed assertion and aborting the program. *-default_also* is an optional flag that, if specified, will execute the default action after executing the *failure-action*; this can be used to print out detailed messages in the *failure-action*, while still aborting the program.

For assertions about (and in) procedures, the value of the input parameters at the time of invocation is useful, because the parameters may be changed during execution of the procedure, and an assertion would no longer have access to the original values. For this, we create variables of the form *param_in*, where *param* is the name of each parameter to the procedure. These **_in* variables are set with the input value of the parameter, and do not change over the execution of the procedure. They can be used in any assertion inside that procedure.

For *assure* assertions, the return value of the procedure must also be available. We provide this through a variable *return_val*. This variable is set in evaluating an *assure*, when the procedure is exiting. While it can potentially clash with an existing variable name (as can the **_in* variables), if the variable is local, no harmful effects will occur since the procedure is exiting, though the original value will not be accessible in the *assure*. The only case where a problem will occur is if *return_val* is a global variable and is declared so in the procedure.

[1]For example, there may be transient states for one of the variables that should be ignored.

General Form of Quantifiers	
quantifier-cmd item-varname[s] data-struct expr	
	A quantifier command evaluates *expr* for all items or item sequences in the data structure (list or array), and returns 1 if the expression is true (nonzero) for the specified quantification over the data structure. If *item-varname* is singular, that variable takes on the value of each item in the list (or index in the array) as the quantified expression is evaluated. If *item-varname* is a list, the variable names in the list take on successive values in the data structure, such that the order of names in the parameter is the order of values in the data structure. If more *item-varname*'s are specified than there are members in the data structure, the quantifier returns 1.
Specific Quantifier Commands	
lall *item[s] list expr*	
	List universal: evaluates *expr* for all items or item sequences, and returns 1 if the expression is true over the whole list, and 0 otherwise.
lexists *item[s] list expr*	
	List existential: evaluates *expr* for all items or item sequences, and returns 1 if the expression is true for any item (item sequence) in the list, and 0 otherwise.
rall *index[es] array-name expr*	
	Array universal: evaluates *expr* for all indices or index sequences, and returns 1 if the expression is true over the whole array, and 0 otherwise. The index list can be prepended with any sorting flags that 'lsort' accepts.
rexists *index[es] array-name expr*	
	Array existential: evaluates *expr* for all indices or index sequences, and returns 1 if the expression is true for any index (index sequence) in the array, and 0 otherwise. The index list can be prepended with any sorting flags that 'lsort' accepts.

Table 2: The four quantifier commands for Tcl.

For this case, we would strongly suggest that *return_val* is not a very good name for a global variable.[2]

2.1 Usage Conventions

Assertions in a program become part of the documentation of the program, its behavior, and its internal interfaces between modules and procedures. This suggests that some of the assertion forms should have standard placements in the program.

The *assert* assertion is a point assertion. It should be used at any point in the Tcl program where the programmer can make a succinct declarative statement about the state of the program, or where they want to ensure that a variable or variables contain the proper data. At the top or bottom of a loop body it can act as a loop invariant.

For procedure interfaces, the *assume* and *assure* assertions should be used. These assertions capture the assumptions that the procedure makes on its parameters and any global variables used, and the assurances of its return values or of any global variables changed. The assertions in effect become part of the declaration of the procedure interface. As such, they should appear as the first statements in the procedure body, after any *global* statements (which are also part of the interface).

The *always* assertion, when it is used as an assertion about global state, should appear at the top of the given Tcl module that has the global state variables. If it is being used as an assertion about the internal state of a procedure, it should appear at the top of the procedure body, after any *assume* and *assure* assertions, or after the variables are created.

3 Quantifier Commands

The two basic aggregate data types in Tcl are the list and array. Assertions may often take the form of describing the properties of a list or an array; for example, specifying that a list contains all positive numeric items. In these cases, universal and existential quantifiers over lists and arrays are useful. Our commands for these take the general form of:

$$\textit{quantifier}\ \{\textit{item}|\textit{index list}\}\ \{\textit{array-name}|\textit{list}\}\ \textit{expression}$$

Our four quantifiers are *lall* and *lexists* for list universal and existential quantification, and *rall* and *rexists* for array quantification. These commands are summarized in Table 2. The first parameter to a quantifier is a list of item names or index names (dependent on whether a list

[2]There are safer ways for naming input parameter and return value variables, such as using an array with named elements, like *assert(return)* for the return value. It was felt that a construct such as this would simply be too cluttering for being able to succinctly read an assertion expression.

Assertion Control Interface	
assertcl *disable ?all|proc|var? ?all|proclist|varlist?*	
	Disables all assertion checking, assertion checking for the given list of procs, or assertion checking for the given list of variables. The keyword *all* can be used in place of a procedure or variable list, indicating all procedures or all variables, respectively.
assertcl *enable ?all|proc|var? ?all|proclist|varlist?*	
	Enables all assertion checking, assertion checking for the given list of procs, or assertion checking for the given list of variables. The keyword *all* can be used in place of a procedure or variable list, indicating all procedures or all variables, respectively.

Table 3: The control interface for AsserTcl.

or array is used). These names are variables that take on successive values of the items in a list (indices into an array). The quantifiers have their general meaning: the expressions in *lall* and *rall* must hold for all sequences of items (indices), and the expressions in *lexists* and *rexists* must hold for at least one sequence of items (indices). A quantifier returns 0 for false and 1 for true.

For example, to specify that all elements of a numeric list are positive, the quantifier

```
lall i $list {$i >= 0}
```

is used. Here the item list is only one item, *i*. One item, however, cannot be used for relations among items, such as specifying that a list is sorted. For this, a quantifier such as

```
lall {i1 i2} $list {$i1 <= $i2}
```

is used, where *i1* and *i2* take on successive values of items in the list. On a list of five items, for example, there would be four pairs of items to compare in the quantifier. If a list is shorter than the number of items asked for, the quantifier returns true (1).

Array quantifiers work the same way, except for one problem—the order of indices to an array is unconstrained,[3] so quantifiers over sequences of elements are less obvious. Still, in many uses of arrays, programmers do have a sequence of indices in mind, and it would not be unreasonable to specify that an array is sorted, like

```
rall {i1 i2} Arr {$Arr($i1) <= $Arr($i2)}
```

For this, our quantifiers process the indices in an *lsort*'ed sequence; that is, we use an index sequence returned by *[lsort [array names Arr]]*, in this example.[4] However, *lsort* by default is an ASCII string sort, and if the indices are numeric, this will give the wrong order. For this, our index list can take any flags that *lsort* takes, and we pass these on to *lsort*. Thus, the above quantifier would be changed to

```
rall {-integer -decreasing i1 i2} Arr \
     {$Arr($i1) <= $Arr($i2)}
```

to specify integer indices in a decreasing order. This syntax is a bit cumbersome and unfortunate, but it is the best we can do.

4 Controlling Assertion Checking

We recognize that assertion checking can be costly in some instances, in terms of performance. This can be especially true with quantifiers over large data structures, and the cost can be somewhat hidden at times. For example, if a procedure is called for each item in a data structure, and that procedure has an assertion about the whole data structure, the assertion effectively turns an $O(n)$ computation into one that is $O(n^2)$.

We expect that when a program is finally released for use, disabling the assertion processing is desirable. Even during system development, as some portions of the system become reliable, it may be desirable to turn off assertion processing for those portions.

In AsserTcl, assertions are controlled through the *assertcl* control interface. Table 3 shows the commands that are allowed by this interface.

Assertion control is provided at the procedure level, variable level (for *always*), and the global level. The global level simply disables all assertion processing, and is effected by issuing the command

```
assertcl disable all
```

for disabling, and with the *enable* keyword for enabling.

By using the *proc* keyword and providing a list of procedure names, assertion disabling can be done per procedure. The command

```
assertcl disable proc P1 P2 P3
```

would disable assertion checking for the three named procedures. Other assertion processing would still take place, provided that global assertion checking had not been disabled. The keyword *all* in place of a list of procedure names disables assertion checking for all procedures.

A similar use of the *var* keyword disables processing any *always* assertions for the specified variables; if a given *always* assertion still relies on other enabled variables, however, it will be processed when those variables

[3] In Tcl, arrays are associative, and any string can be an index into an array.

[4] Of course, a quantifier using only one index does not call *lsort*, since there is no order needed.

change. The keyword *all* in place of a list indicates all variables with *always* assertions.

Even with global-level assertion disabling, an overhead of checking the control variables is still paid. Thus, for a situation such as final release of the software, a *nullassertcl* package that declares empty assertion commands is available. Requiring this package in place of the regular *assertcl* package source leaves just the call of an empty procedure as the only overhead.

Section 7 analyzes the run-time performance of these various levels of assertion checking.

For removing virtually all overhead of using assertions, our future work is expected to include making a program processor that will comment and uncomment assertion commands automatically. This will allow virtually no overhead (except for comment-skipping) to exist in a released program, while still allowing the assertions to remain as documentation, and to be reactivated if needed.

5 Implementing Tcl Assertions

The main Tcl command that enables the addition of assertion commands is the *uplevel* command, which evaluates a script in a different context; thus a procedure can be passed an expression or even a block of statements, and it can evaluate those in the context of its caller, thus making the procedure look like a command in its caller's context.

The other main Tcl feature that we use is the ability to rename an internal command and replace it with our own procedure. We implement a *proc* procedure that acts as a front-end processor to the real *proc* command. Our front-end searches the procedure body for assertion commands, and does the following two things if it finds any.

1. The body of the procedure gets prepended with statements that copy parameter values to a corresponding *param_in* variable. This gives access to the input values regardless of whether the procedure modifies the real parameter variables. Only those *param_in* variables that are used are copied.
2. Each *return* statement in the procedure gets replaced with our own *assertReturn* statement. Ours takes care of evaluating the *assure*'s for that procedure, and then returning from the procedure.

After these two operations on the procedure body are done, we then call the normal *proc* command, so that Tcl registers the procedure. The implementation of *assure* is explained in more detail below.

5.1 Assert, Assume, and Always: The Easy Ones

The *assert* command is implemented in the straightforward manner of a procedure that acts like a command, i.e., *uplevel*'ing the expression to be evaluated and catching any returned exceptions. If the expression itself generates an exception (an error or a user-defined exception), *assert* passes this back to the enclosing context. If the assertion fails, *assert* by default generates its own exception using the *return* command. If the *assert* has an associated *failure-action*, however, that action will be *uplevel*'ed rather than generating an exception.

The *assume* command is really just an alias for the *assert* command, implemented as a procedure that *uplevel*'s an *assert* call. This does not quite fit the definition of an *assume* being evaluated at the start of a procedure invocation, since if the *assume* statement is not placed at the beginning of the procedure body (like we suggest), it will not be evaluated at the start. The alternative would be to preprocess the procedure body and move the *assume*'s, but this would involve some serious syntactic analysis. For now, our decision is to forego this step and rely on conventional placement of the *assume* commands.

For implementing the *always* assertion, the Tcl *trace* command is used, which allows registration of a procedure to call each time a variable is written (reads are also traceable). The *always* assertion creates a uniquely named procedure that evaluates the expression in the assertion and processes any exceptional conditions and a *failure-action*, if there is one. This procedure is then attached to each variable specified in the *varlist* parameter using the *trace* command.

5.2 Assure: The Hard One

Implementing the *assure* command is quite a bit different and more involved. For this command, we do need to modify the procedure body. Due to the *return* command being a not-so-general exception generating mechanism, we do not (and cannot) globally replace the *return* command, but we process the procedure's body and replace each *return* in a procedure with our own *assertReturn* procedure, if the procedure being processed has any *assure*'s. The *assertReturn* procedure evaluates the *assure*'s, and then effects a real *return* from the calling procedure.

Replacing the *return* command can cause some problems and incompatibilities with existing Tcl behavior, but only under well-defined circumstances. The problem stems from the fact that *return* is not simply a command that returns a value from a procedure, but rather a mechanism for generating exceptions. The exception generated by a *return* is passed by Tcl to the context of the caller of the procedure that the return command is in. But when we replace the *return* with our own procedure, we add a layer of procedure call, so that now the context that gets the exception is our caller, not our caller's caller, as it should be. And *return* cannot be *uplevel*'ed, because of the way the call stack is processed in Tcl.

Nevertheless, for Tcl code that simply uses *return* in the normal fashion of returning a value from a procedure,[5] replacing the command with our own does not break any Tcl behavior, and allows us to evaluate any *assure*'s and then effect a real return.

The lexical replacement of *return* with *assertReturn* is not done globally or universally. We only do this in the body of a procedure that is using *assures*, so that any other use of return is not affected. Furthermore, we only replace a return call that begins on its own line (white space excluded). This not only offers simple and quick replacement, avoiding complex syntactic processing, but offers an "out" to a programmer who needs to use return to throw an arbitrary exception in a procedure using *assures*—they can hide the return from our processing by using a construct such as

```
if 1 { return -code $Exception ... }
```

Since this return does not begin its own line, our replacement method will ignore it.

The *error* command can also be used to return from a procedure, but since this method is not interceptable (due to the above-mentioned limitations on the *return* command), we make no effort to catch this command. In [4], it is recommended that *error* not be used except for true errors, in any case.

6 Examples

This section presents a few simple examples to give a flavor for what assertions in Tcl look like and can do.

A point assertion about variables x and y:

```
assert {$x>4 && $x<$y+2}
```

A point assertion that simply prints a warning and does not abort the program:

```
assert {$x>4 && $x<$y+2} {
   puts "Assert failed: x:$x y:$y"
}
```

An assertion about some global variables over the life of the program:

```
always NumUsers {$NumUsers >= 1 && \
                 $NumConnected > $NumUsers}
```

Note that this assertion only gets checked when $NumUsers$ changes, not when $NumConnected$ changes.

A simple procedure declaration with interface assertions:

```
proc square {x} {
   assume {$x+1 > $x}  ;# tests for numeric value
   assure {$return_val == $x_in * $x_in}
   set x [expr $x * $x]
   return $x
}
```

[5] A procedure return "exception" is the only one that can be passed up one extra level.

A procedure that returns the quotient and remainder of a divide operation:

```
#
# divmod: returns a two-element list of quotient
#         and remainder
#
proc divmod {dividend divisor} {
   # simply test for numeric value
   assume {$dividend + 1 > $dividend}
   # also, make sure non-zero divisor
   assume {$divisor + 1 != 1}
   # must return the correct quotient and remainder
   assure {[lindex $return_val 0]*$divisor_in +  \
           [lindex $return_val 1] == $dividend_in}
   # procedure body
   set quot [expr $dividend / $divisor]
   set rem  [expr $dividend % $divisor]
   return [list $quot $rem]
}
```

Assert that all list elements are positive:

```
assert {[lall item $list {$item>=0}]}
```

Assert that a list contains an element "Jon":

```
assert {[lexists item $list {"Jon" == $item}]}
```

Assert that a list is sorted:

```
assert {[lall {i1 i2} $list {$i1 <= $i2}]}
```

Assert that an array contains the value 42:

```
assert {[rexists i Arr {$Arr($i) == 42}]}
```

Assert that an integer-indexed array is sorted:

```
assert {[rall {-integer i1 i2} Arr \
       {$Arr($i1) <= $Arr($i2)}]}
```

Of course, an arbitrary Tcl procedure can be called in an assertion (though it should not have side effects!), so a procedure that checks the consistency of a more complex data structure can be used, like:

```
assert {[lall cust $CustomerList \
       {[CheckCustRecord $cust]} {
   puts "Warning: Customer is malformed -- $cust"
}
```

which asserts that a list of customer records are all consistent, but only prints a warning if it is not.

7 Performance Evaluation

While assertions serve as unambiguous, written-down, declarative specifications of a program behavior, their compelling reason of existence is that they can be evaluated at run-time, providing increased assurance that a program's behavior is correct.

Run-time evaluation of assertions does not come for free, of course. They add extra processing to a program and can add significant increases in program execution time. For example, in the case of an assertion using *lall*, the expression must be evaluated over the whole list to evaluate the assertion. If it is in a loop, each iteration

Assertion Configuration	Tcl 7.6	Tcl 8.0a2
full assertions	97.53	48.16
-P1	79.43	39.32
-P2	78.12	38.64
-P1,P2,P4	62.17	28.51
-P3	35.91	23.63
all disabled	3.69	2.87
-P1-8	2.56	2.29
nullpackage	1.14	0.73
comment-out	0.95	0.66

Table 4: Performance of assertions on Tcl 7.6 and Tcl 8.0a2

of the loop processes the whole list, potentially making an $O(n)$ operation become $O(n^2)$.

This overhead can be controlled during development and testing through the judicious enabling and disabling of assertions in procedures and variables, as explained in Section 4.

Here we present some performance numbers for a small Tcl program developed using assertions. The program was a non-interactive, stochastic Petri net simulator, and made heavy use of arrays. The program consisted of thirteen procedures, all of which had assertions in them. Two procedures used the *lall* quantifier in their assertions. The program did not contain any *always* assertions. The program was 177 source lines, excluding comments and brace-only lines, and had 16 assertions.

We ran the program using both Tcl 7.6 and Tcl 8.0a2, on an UltraSparc running Solaris. Table 4 shows the performance of a Tcl program over various configurations of assertion checking. Each value is the mean of 5 runs, timed using the Unix *time* command. Rows with '-P#' configurations have assertion checking disabled for the specified procedures, but enabled for everything else. The last line in the table was the running time of the program with all assertion statements hand-commented out of the program. This represents the lower bound on running time.

As can be seen, full assertion checking carries a high price for this particular example. It runs almost 100 times slower on Tcl 7.6, and almost 75 times slower on Tcl 8.0. Disabling various procedures reduces the time by considerable amount, with the disabling of assertions in P3 cutting the execution time by over half on Tcl 8.0, and almost two-thirds on Tcl 7.6.

The two methods of disabling assertion checking, using *assertcl disable all* or using the *nullassertcl* package both show good performance relative to the commented-out performance. The *nullassertcl* package performance is within 20% on Tcl 7.6 and within 11% on Tcl 8.0, indicating that it is deployable technology—with an interactive Tk program, an 11% overhead would not be noticeable in most instances.

An interesting note is that disabling assertions in eight of the thirteen procedures results in performance that is faster than the global *assertcl disable all*; this happens because the procedure disabling actually skips inserting any assertion processing in the body of the disabled procedure(s). Thus, if the time-critical procedures are disabled in this manner, the other procedures can still check assertions, and the performance remains good.

For the list quantifiers, allowing the commands to use list names rather than lists would improve their processing time immensely.[6] However, this usage would not be consistent with the normal Tcl list commands, so we chose to leave the calling interface as using a list. However, we may decide to change this, or support both, in the future.

8 Conclusion

In this paper we presented AsserTcl, an assertion package for the Tcl programming language. We feel that assertions will make the development of robust systems easier. At the same time, we hope to use this foundation for future research exploring more ideas in assertions for programming languages, and for evaluating the effectiveness of assertions.

Tcl is an evolving language, and the next version will have *namespaces*, which will be akin to packages or modules. Earlier work on Ada packages [2] and Eiffel objects [3] have shown that modules present interesting issues in developing effective assertions for them. In addition, object oriented extensions to Tcl exist, and these can provide a platform for exploring issues about assertions for classes and objects.

In developing this package, some limitations were encountered that could be ameliorated by adding some functionality to the Tcl core language. Our suggestions are:

1. Extend the exception generating capability of the *return* command so that it is possible to throw an exception to a specified level, rather than always the caller. This would enable the true replacement of the *return* command (enabling other debugging packages), and would generalize the exception mechanism.

2. Extend the *info* command with access to a procedure's current line number (when executing) and file name. This is also a feature that would enable more functional debugging packages. The *info level* command could be extended to provide the line number currently being executed in that level, and a command such as *info proc procName* could be added that would return the file name and the beginning line number of the procedure definition.

[6] A simple test with a 20-element list showed an order of magnitude in speed difference.

Availability

AsserTcl is, and hopefully will remain, a Tcl-only package, to ensure the best portability and ease of installation and use. It is freely available at http://www.cs.colorado.edu/~jcook/TclTk.

Acknowledgements

I would like to thank Professor Mikhail Auguston for discussing aspects of assertions for programming languages, and to Professor David Rosenblum for first turning me on to assertions. Many thanks to the anonymous reviewers of this paper as well. Their suggestions improved it greatly.

REFERENCES

[1] M. Auguston, S. Banerjee, M. Mamnani, G. Nabi, J. Reinfelds, U. Sarkans, and I. Strnad. A Debugger and Assertion Checker for the Awk Programming Language. In *Proc. International Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, 1996.

[2] David Luckham. *Programming with Specifications: An introduction to Anna, a language for specifying Ada programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1990.

[3] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, 1988.

[4] John K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[5] David S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 1995.