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

# Using Tcl to Script CORBA Interactions in a Distributed System

Michael L. Miller
*Advanced Micro Devices*
Srikumar Kareti
*Honeywell Technology Center*

For more information about USENIX Association contact:

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

# Using Tcl to Script CORBA Interactions in a Distributed System

Michael L. Miller

[a]*Advanced Micro Devices, MS 608, 5204 E. Ben White Blvd., Austin, TX 78741*

Srikumar Kareti

[b]*Honeywell Technology Center, 3660 Technology Dr., MN 65-2600, Minneapolis, MN 55429*

## ABSTRACT

In this paper we present the extensive use of a scripting language (Tcl) to run human readable/editable scripts in a CORBA distributed batch environment. The developed system is the "Advanced Process Control Framework" as outlined in the "APC Framework Initiative" which is a research and development project undertaken by AMD and Honeywell under the support of the U.S. Department of Commerce, National Institute of Standards and Technology. The APC Framework System has been deployed in AMD's fab 25 and is fully functional. The paper discusses the issues involved in developing scripting mechanism which is capable of both interacting with other CORBA components and handling various data structures which are otherwise not addressable by the underlying scripting language. The flexibility and extendibility of the Tcl scripting language makes it easy to extend the core language. The paper also establishes the need for thread-safeness of Tcl. Examples of various data manipulation operations, calls to different CORBA components, and calls that help in synchronization are discussed.

## 1. BACKGROUND

In order to fully describe how Tcl is being used in our CORBA environment, it is first necessary to give some background into the project and the distributed system that we are developing. Starting in 1994, AMD identified the need for an extension to our current Manufacturing Execution System (MES) that would support deployment of Advanced Process Control applications quickly and easily into our semiconductor manufacturing facilities (fabs). AMD internally developed a functional specification for such a system, which was completed in mid-1995. The National Institute for Standards and Technology (NIST), a section of the Department of Commerce, announced shortly thereafter an Advanced Technology Program (ATP) competition. Under this ATP, NIST would cost-share up to 49% of a research and development project that would further development in the area of MES systems and integration. The NIST program provides multi-year cost-share funding to industry-led joint ventures to pursue research and development (R&D) projects with high-payoff potential for the nation. Its goal is to accelerate technologies that are unlikely to be developed in time to compete in rapidly changing world markets without such a partnership between industry and the Federal government.

AMD, in partnership with Honeywell, a leading control systems supplier, and SEMATECH, the consortium of US semiconductor manufacturers, proposed the APC Framework Initiative (APCFI).

### 1.1 APCFI Project

The goals of the Advanced Process Control Framework Initiative project (APCFI) were outlined in the program proposal presented to NIST (National Institute for Standards and Technology) in October, 1995. These goals were to: enable effective integration

Further author information -

[a] Email: michael.miller@amd.com; Telephone: 512-602-3959; Fax: 512-602-5299

[b] Email: skareti@htc.honeywell.com; Telephone: 612-951-7302; Fax: 612-951-7438

of "Advanced Process Control" applications into a semiconductor fab to improve manufacturing capital productivity, product consistency, and product yields; establish integration technology for multi-supplier "Plug-and-play" APC applications; and to demonstrate commercial viability of the APC Framework and its components. To sum up, the main goal of the APCFI projects was to develop a system that would significantly reduce the time, cost, and integration efforts needed to deploy APC solutions.

The scope of the APCFI projects includes support for Feedforward and Feedback Run-to-Run control and Fault Detection applications spanning multiple processes and fab tools and utilizing 3rd-party control software, such as Modelware®, Matlab®, Matlab Toolkits, Mathematica®, and LabView®.

In order to validate the design and implementation of the APC Framework, a number of control projects were selected for early deployment into one of AMD's semiconductor fabs using initial versions of the APC Framework.

### 1.2 CORBA

CORBA (Common Object Request Broker Architecture) is a specification of an "architecture for an open software bus on which object components written by different vendors can interoperate across networks and operating systems" (Orfali et. al., 1996). It is used by the APC Framework to allow the distributed components of the framework to communicate. In specific we used Orbix, IONA's implementation of CORBA to develop APC.

### 1.3 Overview of the APC Framework

The APC Framework has been designed to work along with a fab's MES (Manufacturing Execution System – in AMD's case, this it WorkStream by Concilium) and CEIs (Configurable Equipment Interface) to provide APC functionality. It is composed of not one large program, but a number of smaller, specialized pieces that work together. The "interchangeable parts" of the APC Framework are called components. These components are analogous to stereo components, where each component is

1) An independently running entity
2) Provides a subset of the overall APC Framework functionality
3) May be provided by a different vendor

The APC Framework standard describes the functionality, interface, and behavior of each component. The central component at run-time is the Plan Execution Manager, which utilizes Tcl and is described in the next section.

## 2. THE PLAN EXECUTION MANAGER

To support the goals of the APCFI project, it was necessary to develop a system that would be flexible enough to support just about any supervisory-level

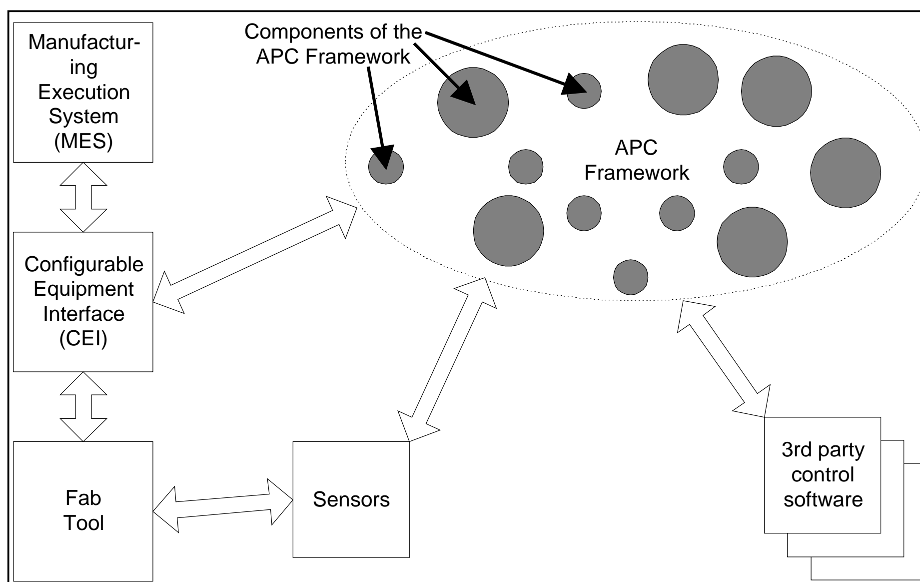


**Figure 1: The relationship of the APC Framework to other software systems.**

(e.g. Run-to-Run) control application. To this end, rather than trying to pre-define a generic sequence of system interactions that would be used at run-time, the project chose to use a scripting approach to allow the maximum flexibility in how the system was used.

The Plan Execution Manager (PEM or PE) component is the "choreographer" of the APC Framework. It is in charge of doing "APC" at runtime for a particular process or metrology tool. To do this, it has the ability to access all of the capabilities of each of the other components in the APC Framework. It executes, or interprets, APC Plans (a collection of Tcl scripts), which specify the actions to be taken before and/or after a lot is processed on a fab tool. The Tcl scripts define not only what the APC Framework does, but in what order they are carried out.

### 2.1 Why Tcl?

When the project needed to define the scripting language that would be used to drive the system activities at run-time, a number of possibilities were evaluated. Tcl, Perl, and Python were examined, as well as developing a custom language. The latter was quickly dropped because of the rich set of well tested scripting languages readily available and the fact that project resources could be better spent developing other functionality in the system.

Some of the comparison criterion used were: ease-of-use, ability to embed in a C++ application, extensibility, support for CORBA, and support for C structures. The first criterion, ease-of-use, was perhaps the most important. Since the script writers would be programming novices, ease-of-use and a shallow learning curve were critical to the project's success among the users. As the scripting language was to be embedded into a CORBA C++ server, the chosen scripting language had to be able to be embedded into a C++ application. Extensibility was an important consideration, since the chosen language was unlikely to have all of the functionality that would be required. Finally, since the scripts would need to access CORBA functionality and manipulate data in C-like structures, the ability of the language to accommodate these was important.

Python, while it had many features that lent itself to use in an object-oriented environment such as was being defined for the APC Framework, also carried some drawbacks. Foremost amongst these drawbacks was the fact that the scripting language itself is very object-oriented. The users that would become the main users of this scripting language were not object-savvy, and so Python represented a language that would require much more up-front learning on the part of the users before it could be used effectively. The basic Python language, however, did have some support for more sophisticated C & C++ data structures, and was well-suited for use as an embedded language in a C++ application. Python also seemed extensible, although some of the overhead of reference counting, etc., would add some additional work to any extensions being built. Finally, Python did not have any support for CORBA communications in its core, but there was some work ongoing in this area by others.

Perl also had good facilities to support custom user extensions to the language. However, it suffered as well in ease-of-use – the reviewers felt that Perl was syntactically more complex than some other languages, like Tcl, and hence would be more difficult to learn by the script writers. Perl also lacks key facilities which make its use in an embedded application much more complex than either Tcl or Python. Finally, even though Perl seemed to have sufficient capabilities to support some of the more complex data structures that the APC Framework would use, it lacked any CORBA capabilities.

Tcl had an advantage over both Perl and Python in its simpler syntax and hence easier learning curve for new users. Even better was its support for use as an embedded interpreter in a C/C++ application. Since it was developed from the start for use as an embedded language, its facilities were better than both Perl and Python. While there was not any support for CORBA or data structures per-se in the Tcl core language, there were already-developed extensions that would provide this functionality. Even though the project made the decision not to use these extensions in favor of creating our own, the fact that some form of this capability existed already made it easier for the project to roll our own. Overall, Tcl's ease-of-use and embeddability made it the best choice for the APC Framework.

### 2.2 Use of Tcl in the Plan Execution Manager Component

When the fab process or metrology machine informs the PEM that a specific lot has been brought in for processing, the PEM pulls up a Plan Executor (PE) object to execute the "APC Plan" for that run. The Plan Executor runs various scripts designed by the "Process Engineer" (a.k.a. script writer) and feeds back correction information to the machine to help maintain consistent performance of the machine. The

process engineer is typically a chemical engineer and hence it greatly helps to have the scripting language to be readable and English like. There are three types of Scripts: Main Scripts, Sub Scripts, and Event Scripts. These scripts are used in an APC application to define the sequence of actions that the APC Framework performs. The scripts are bundled together in an APC Plan: one (and only one) Main Script, zero or more Sub Scripts, and zero or more Event Scripts. The Main Script is used like the `main` function in C – it is the first script run by the system when it executes a Plan. The subscripts are used to define procedures that the main or event scripts may use. The event scripts are executed in response to certain events should they happen in the system.

When the PE is called to execute a Plan, it begins by creating a Tcl interpreter for the main script in a new process thread. This interpreter is initialized and the APC extension loaded. Next, the PE defines all of the procedures by evaluating all of the subscript files (using `Tcl_EvalFile`), one at a time. These subscripts contain routines common to all the scripts. Finally, the PE executes the main script, one line/command at a time. This is done so that the PE can respond to other requests that may interrupt the execution of the main script between execution of each Tcl command. We allow the user to be able to write the command in more than one line as long as he maintains "Tcl-like" syntax.

While the main script is executing, the PE may receive notification of certain events happening in the system. If there is an event script defined for that event, the PE will execute it in a manner similar to the execution of main scripts. Each of the main script and the Event scripts has an interpreter of their own. Each Interpreter runs in its own process thread and can communicate via shared data and mutex-like locks. This clearly marks the need for a thread safe scripting tool. In the previous versions of Tcl, we were forced to use simple mutex locks around the Tcl library, rather than spending the time modifying the Tcl core to be thread-safe. The latest version of Tcl (8.1) promises to be thread safe, which will make the use of multiple interpreters in separate process threads much easier to use. This is a performance gain for the APCFI system because the locks we used to make Tcl thread safe were very coarse grain. It is not very uncommon to have about five plans running at a time, each in parallel and each of the PEs having a main script and multiple event scripts all running in parallel. The need for thread safeness was high enough to consider porting ptTcl from unix to NT, but the eminent release of Tcl 8.1 made this unnecessary.

When the main script completes, all of the Tcl interpreters are deleted.

## 3. TCL EXTENTIONS FOR APC

Even though Tcl was chosen to be the scripting language used in the PEM component, in its basic form it did not have all of the functionality needed by this project. Among the added functionality was: Tcl scripts needed to be able to be run in parallel (1 main, multiple sub and event scripts); they needed to communicate data and synchronization information with each other; these scripts needed to create, understand and interpret complex data structures; the scripts needed to communicate to the rest of the world via CORBA; and, finally, since some setup tasks (CORBA calls) might take a considerable amount of time, there was a need to include the capability to run such tasks in the "background" (another Tcl interpreter run in another process thread) and let the calling script continue until it was in need of the data from the background task. Existing Tcl extensions, along with the possibility of building our own, were evaluated. The extensions/modifications to Tcl that supported all of this functionality is discussed below.

### 3.1 Complications using Tcl

In its core form, Tcl utilizes all data in the form of strings. This makes life simple for the scriptwriter, but causes complications when trying to interoperate in a distributed CORBA environment that uses more complex data structures. Extensions do exist for creating/handling other data structures, but they lacked the ability to handle the CORBA data types that we required. Also, these extensions were built to allow the script writer to construct new data structures "on the fly". While this is desirable in the general sense, the APC Framework utilizes a fixed set of data structures, so this added flexibility is not needed and in fact adds to the learning curve for the script writer. It is for these reasons that the decision was made to write a fixed set of new commands from scratch, which had the added advantage of being able to incorporate features that would allow the data structures to be shared between scripts running in separate interpreters. Our extensions are based on using the `Tcl_SetAssocData` and `Tcl_GetAssocData` calls to store and retrieve the data structures when needed.

In addition, the Tcl core does not have the ability to perform CORBA invocations. We were aware of initial developments extending Tcl to allow CORBA calls, but these packages were either not on the needed version of Tcl or on the necessary platform. Also, as was the case with the data structure extensions, the CORBA extensions provided general facilities for constructing and making a CORBA call to a server. While this provides more capability to the script writer, it carries with it a high price in terms of script complexity. Again, the set of CORBA methods that the script would need to access would be finite and fixed, so this type of general CORBA capabilities was not needed. Finally, we wanted to have a higher level of abstraction where we would make multiple CORBA calls in the same Tcl command rather than have one call to each CORBA invocation. By writing our own CORBA extension specifically for this project, we had that flexibility.

Finally, Tcl has no simple mechanism to support communication between Tcl interpreters running in different threads. In fact, until recently the Tcl core itself was not thread-safe. ptTcl, a multi-threaded version of Tcl, was available for Sun Solaris. It provided not only the ability to launch multiple interpreters in separate threads, but to also communicate with those separate interpreters, However, we did not try to port it to NT due to lack of time and resources. Instead, we built into our extensions the ability to use mutex locks between interpreters and to copy data into and out of a common memory space.

### 3.2 Data object-related commands

The first additions to the Tcl language made by this project were commands that give the script writer the ability to create, manipulate, and delete all of the different data structures/objects that the APC Framework uses. In general, there are two types of data objects: pure structures and sequences of structures. One new command was defined for each

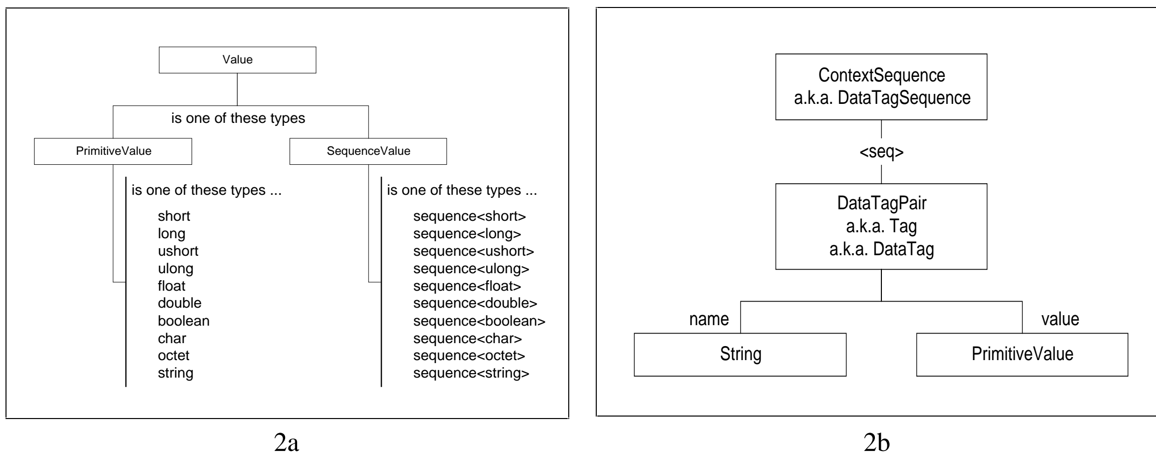


**Figure 2: Value (a), DataTagPair (b) and DataTagSequence (b) data structures**

| Command | Arguments | Returns | Comments |
|---|---|---|---|
| DTPair | | | |
| create | Name Type value DTPkey | | Create DataTagPair |
| get | DTPkey | Name [Prim./ Seq.] Type value | Return the contents of DataTagPair |
| getvaluekey | DTPkey Vkey | | Put the contents of the DataTagPairs's PrimitiveValue in a new Value called Vkey |
| set | Name Type value DTPkey | | Set contents of existing DataTagPair |
| delete | DTPkey | | Delete the DataTagPair |

**Table 1: The DTPair Command**

| Command | Arguments | Returns | Comments |
|---|---|---|---|
| DTSeq | | | |
| create | <DTPkey> DTSkey | | Create a DataTagSequence using a list of DataTagPair keys |
| createbyvalues | <Name Type value> DTSkey | | Create a DataTagSequence using a list of contents of DataTagPairs |
| length | DTSkey | Length | Return the number of elements |
| names | DTSkey | <names> | Get all names of DataTagPairs in the DataTagSequence |
| geti | Index DTSkey DTPkey | Name [Prim./ Seq.] Type value | Return the contents of DataTagPair at index |
| getdtpairkey | Name DTSkey | | Create a new DataTagPair using the element at the index |
| getvalue | Name DTSkey | Name [Prim./ Seq.] Type value | Return the contents of the DataTagPair from DataTagSequence with Name |
| getvaluekey | DTPkey Vkey | | Same as above but store the Value against the key |
| add | DTSkey | | Add a DataTagPair to the sequence |
| remove | [Index \| Name] DTSkey | | Remove the DataTagPair at index I or with name N and store it back |
| delete | DTSkey | | |

**Table 2: DTSeq (DataTagSequence) command**

data object type. This command uses the first argument as a switch to define what to do with that object: in general, to create it, get its contents, set its contents, and delete it. A particular instance of a data object is referenced by a unique name – like a variable name. This name, or key, is passed as an argument to each of the new commands.

Figure 2 shows an example of some of the data objects used by the APC Framework – the Value, DataTagPair, and DataTagSequence structures. Tables 1 and 2 list the commands used to access the DataTagPair and DataTagSequence as an example.

**Example code to pass non-string variables in Tcl**

```
DTPair create "Length" "long" 50
    testTag1

DTPair create "Width" "float"
    25.5 testTag2

DTSeq create "testTag1 testTag2"
    testDTSeq

set n [DTSeq length testDTSeq]

puts "Length of DTSeq (should be
    2): $n"
```

The first two lines create “DataTagPairs” with keys (variable names) ‘testTag1’ and ‘testTag2’. Line 3 recalls from memory the ‘testTag1’ and ‘testTag2’ DataTagPairs and forms the DataTagSequence with the tag ‘testDTSeq’. Lines 4 and 5 show some simple steps to do an operation on a data structure and display the results; in this case, get the length (number of elements) of the DataTagSequence and print it out.

**Infrastructure for accessing different data structures**

APC has a well defined hierarchy of well defined data types. To be able to handle all these different types of data structures, we have a global map (an object that keeps a list of names and associated data) for each of the different data types. When create operation is called for any particular data type, the created data type is stored in the map against the name/key passed in as an argument. The key is then used in any routine to pull the data from the map. The mechanism is local to each script executor and there is no naming conflict across scripts in the same plan. Also, each data structure has its own map and hence no naming conflict exists across different data types. Finally, to support sharing data between scripts in the same APC

Plan, a plan-level map is used, and each script can copy data to and from that map.

### 3.3 CORBA method invocation commands

The second major category of new Tcl commands provides the script writer with the ability to invoke methods on other components of the APC Framework via CORBA. In a manner similar to the way data object commands were defined, one command per IDL interface was created. Each command uses the first argument as a switch to select what functionality of that component will be accessed. In general, one switch was defined for each logical interaction. These logical interactions were defined at the granularity that a script writer would need, and no finer. In some cases, there is a one-to-one correspondence between command + option and CORBA method, and in other cases many CORBA calls are wrapped together in one option.

Table 3 below shows an example command – this command handles interactions with the DataStore component. This is a good example where one option, for instance '`store`', results in multiple CORBA calls. In the case of '`store`', the script uses the 'store' option to put data into a database in a specific way. The C++ implementation of that command first invokes a 'find' command on the Data Store component to find if there is any similar data already stored in the database. If there is data already there, then the command uses a second CORBA method to replace the existing data with the new data. On the other hand, if nothing appropriate is in the database, the command uses an alternative CORBA method on the Data Store to create a new data set in the database.

By combining CORBA method invocations into a logical function, the extended scripting language can be kept relatively simple. The script writer doesn't deal directly with CORBA methods, just functionality that he/she needs to use.

### 3.4 Miscellaneous commands

In addition to commands to manipulate data objects and invoke methods on distributed objects, other utility commands were added. One need was for synchronization and communication between the main and event scripts. For synchronization the PE uses simple mutex locks – one script can set a lock and wait for another script (running in another process thread) to release the lock. In addition, the PE controls a global memory area that is separate from the memory used by each of the scripts. In order to exchange data, the scripts use this global memory through a new command which was added to allow the scripts to copy data to or from this global memory.

The example below illustrates two scripts using locks to synchronize their activities.

**Example scripts**

<main script>

```
Lock create AlarmEventLock
Lock wait AlarmEventLock 180
if([Lock status]) {
# event received
...
} else {
```

| Command/Option | Arguments | Returns | Description |
|---|---|---|---|
| `DataStore` | | | |
| `store` | `[temp | perm] DTSKey NVSKey` | | stores data in DataStore |
| `retrieve` | `[temp | perm] DTSKey [N | NVSKey]` | `{[Prim./ Seq.] Type value}` | retrieves only the exact matching data from the DataStore |
| `query` | `[temp | perm] DTSK` | `"contents of stored data"` | returns all partially-matching data stored in the DataStore |
| `delete` | `[temp | perm] DTSK` | | deletes data stored in the DataStore |
| | | | |

**Table 3: Example CORBA method invocation command from the APC Framework**

```
# timed out
...
}
```

<event script>

```
# do some processing
Lock unlock AlarmEventLock
```

In this example, the main script needs to wait until the event script has run past a certain point before it continues executing. To accomplish this, the main script creates a lock called 'AlarmEventLock' and then waits for it to be released. When the event script reaches the `Lock unlock` command, it releases the 'AlarmEventLock', which allows the main script to continue. In case the event script never releases the lock for whatever reason, the main script times out after 180 seconds and continues with the rest of the script.

In some cases, there are time-consuming activities that the scripts need to perform. These time-consuming commands typically contain CORBA calls, but in general can be any activity. In order to provide potential performance improvements, these time-consuming activities can be performed in the "background" for the cases where the command needs to be completed before some point is reached, but not necessarily in a deterministic order. A new command was written that allows a script writer to execute a Tcl command in a separate Tcl interpreter. This new interpreter is created and executed in a separate process thread from the calling script. This gives the calling script the ability to continue executing, while the separate interpreter handles the slower activities. Only a single command is allowed to be executed in this manner, but that command can be a procedure call, so just about any activities can be performed. This mechanism is ideal for slow setup processes and database access.

**Example code**

<sub-script>

```
proc setup_plugin {name} {
    set ex_plugin [PlugIn Setup
    $name]
    Move global ex_plugin
    ex_plugin_global
}
```

<main script>

```
set fork_wait [Fork setup_plugin
   "example_plugin"]
# do other things
...
# now wait for the background
task to finish
#(if it hasn't already)
Lock wait $fork_wait
```

The command `Fork` runs "*setup_plugin "example_plugin"*" in the separate Tcl shell. Prior to executing the command in the separate Tcl interpreter, the subscript file is evaluated in order to define any needed subroutines. The command also returns the name of the lock which will be released once the command has completed execution.

## 4. SUMMARY AND CONCLUSIONS

Tcl has proven to be a great base language upon which to build a CORBA scripting language. While the fact that in general Tcl deals only in strings may seem to hamper its use in such a distributed environment, the addition of specific commands that deal with the data types of interest keeps the command syntax and scripting simple and easy to learn.

The biggest drawback of embedding Tcl in such a multi-threaded component as the PE is its lack of thread-safety, which has begun to be addressed, eliminating the need for extensive modification of the Tcl core or locks around the library calls.

While this project started with Tcl 7.6, we have kept up to date with the recent Tcl releases. However, we have not been able to rewrite the extensions to utilize the object interfaces added in Tcl 8.0, so much improvements could be made to the APC extensions.


## ACKNOWLEDGEMENTS

Portions of this work was performed under the support of the US Department of Commerce, National Institute of Standards and Technology.

## REFERENCES

1. Orfali, R. , Harkey, D., Edwards, J., The Essential Distributed Objects Survival Guide, (Wiley, 1996).