PEM - Procedure Execution Manager

Claude Saunders
Operations Analysis Group
Accelerator Systems Division
Argonne National Laboratory
September 25, 1996

1. Introduction

This document serves as a guide and reference for all aspects of using the PEM as well as tcl-based machine procedures at the APS in general. The PEM application is introduced first, along with a brief summary of the structure of machine procedures and the various ways in which they may be used. The next section gives detailed examples of tcl-based machine procedures. Lastly, a reference is provided on the utility procedures available for writing a machine procedure, and a guide for installing the PEM. Note: the PEM, its utility procedures, and machine procedures all depend on the APS Tcl/Tk Library (http://www.aps.anl.gov/asd/oag/oagSoftware.html). The PEM also depends on some scripts and inetd.conf entries being set up in advance. See the reference section for this.

1.1. PEM

The Procedure Execution Manager is a graphical interface which permits the user to invoke machine procedures and monitor their execution. Simply type "pem &" to start it. Machine procedures are tcl procedures which follow a particular format. These machine procedures are put in one or more files, and the PEM uses tcl autoloading to access them. The PEM accepts as input a configuration file (pemConfig in the current directory by default, or specified with the -pemConfig command line option). This file lists the various machine procedures which are available, and provides default arguments to those procedures. An example pemConfig file is as follows, one line per procedure:

```
APSMpPARTurnon -mode inj -energy 400 -standardizationTime 12 -restoreFile res.gz
APSMpPARStandardize -energy 400 -standardizationTime 2 -restoreFile res.gz
```

In a separate tcl library file (*.tcl), the machine procedures APSMpPARTurnon and APSMpPARStandardize are defined. If this library file is installed via the OAG software repository, access is automatic. If you are testing procedures locally, make sure you have the library file and tclIndex in the directory you invoke "pem" from.

The PEM allows you to select a host machine to run the procedure on (defaults to current host). If you choose another host, make sure you have xhost permission (i.e. “xhost + hostmachine”). You may also select an execution mode: Automatic, Semi-Automatic, or Manual. This defines the degree of interaction and monitoring that will occur. The mode determines what happens when a “step” is encountered in your procedure. In Manual mode, the PEM will stop at each “step” until you press continue. In Automatic mode, all steps are ignored. Semi-Automatic mode only stops on the first “step”.

1.2. Machine Procedures

Machine procedures are tcl procedures which follow a particular format (i.e. you are required to utilize certain utility procedures within). The simplest machine procedure is as follows:

```
proc    APSMpDoThis {args} {
    set opt1 default1
    set opt2 default2
    APSParseArguments {opt1 opt2}

    # Do something
    set fd [open /tmp/stuff w]
    puts $fd "You supplied $opt1 for opt1 and $opt2 for opt2"
    close $fd

    APSMpReturn ok "arbitrary return data here"
}

# Note: a companion "description" procedure should be defined.
proc    APSMpDoThisInfo {} {
    return "This procedure should return a string describing the machine proc."
}
```

This procedure just takes two optional arguments and writes their value to a file. Note that all arguments should be passed using the APSParseArguments calling convention (see the APS Tcl/Tk Library document at http://www.aps.anl.gov/asd/oag/oagSoftware.html). Lastly, the APSMpReturn call must be made as the last executed statement of the machine procedure. This is true even if you return prematurely due to some detected error condition.

There are other utility procedures you may call from within your machine procedure. These will be covered in the next section, but here’s a list:

```
    APSMpStep          Defines steps within a procedure
    APSMpInterface     Returns user-interface execution context (0 or 1)
    APSMpParallel      Schedules execution of a procedure in parallel
    APSMpJoin          Synchronizes current procedure with parallel procedure
    APSMpAbort         Abort execution of selected parallel procedure
```

1.3. Using Machine Procedures

Machine procedures may be executed in different contexts. They are designed to be executed primarily from within the PEM. The PEM takes full advantage of the format of a procedure to permit monitoring and controlling execution.

Machine procedures may also be executed from any arbitrary tcl script or from within another machine procedure. Special “dummy” versions of APSMpReturn and APSMpStep will be loaded from the APS Tcl/Tk Library. These dummy versions essentially do nothing, but allow your machine procedure to be executed as if it were a regular tcl procedure. APSMpParallel and APSMpJoin may be used at any time, provided you have configured inetd and installed some scripts (reference).

Lastly, machine procedures may be executed from the command line by using the command line utility “runProc”. This script starts up a tcl interpreter and evaluates the command line arguments in the interpreter. Ex:

```
runProc APSMpPARStandardize -energy 400 -standardizationTime 2 -restoreFile
res.gz
```

2. Example Machine Procedures

We begin with a simplified version of the APSMpPARStandardize procedure. This will make full use of APSMpStep to permit interaction via the PEM.

```
proc    APSMpPARStandardize {args} {
    set controlFilesDir /home/helios/oagData/controlFiles
    set SnapshotDir /home/helios/oagData/SCR/snapshots
    set status ok

    set energy 400
    set restoreFile ""
    APSParseArguments {energy restoreFile}

    # Note: by convention we define an initial step. If this procedure is
    #       being executed in Semi-Automatic or Manual mode, the procedure
    #       will stop here and bring up the InitDialog, giving the user an
    #       opportunity to change the default arguments interactively. In
    #       fact, the dialog procedure may set any variable in the scope of
    #       this procedure, provided it is passed using the -vars option. The
    #       InitDialog is defined below.
    APSMpStep init -promptDialog APSMpPARStandardizeInitDialog -vars {energy \
        restoreFile}

    if {![string length $restoreFile]} {
        # Note: Info window will only be brought up here if this procedure
        #       is being executed in a context which provides a tk interpreter,
        #       and the procedure is not being executed in Automatic mode.
        if [APSMpInterface] {
            APSInfoWindow .parStandAlert -name "Warning" -infoMessage \
                "No Restore File Given, returning" -modal 1
        }
        APSMpReturn error "No restore file given"
    }

    # Note: here there is no dialog associated with this step.
    APSMpStep standardize
    set sfile ${controlFilesDir}/PAR/P${energy}MeV.std
    if [catch {exec standardize $sfile} result] {
        if [APSMpInterface] {
            APSInfoWindow .parStandAlert -name "Warning" -infoMessage $result \
                -modal 1
        }
        APSMpReturn error "unable to exec standardize"
    }

    # Note: this procedure will block here until standardization is complete. If,
    #       for example, you are running PARStandardize from a higher level
    #       procedure, you may wish to use APSMpParallel to execute
    #       PARStandardize. That way, your higher level procedure can continue
    #       doing work while this procedure sits and waits for standardization
    #       to complete.
    APSMpStep waitForStandardization
    while {[exec cavget -list=P:DCPS:CondCALC]} {
        update idletasks
```

```
        after 5000
    }

    APSMpStep restoreMagnets
    if [catch {exec burtwb -f $restoreFile} result] {
        if [APSMpInterface] {
            APSInfoWindow .parRestoreAlert -name "Warning" -infoMessage $result\
                -modal 1
        }
        APSMpReturn error "unable to exec burtwb"
    }

    APSMpReturn ok "success!"
}

# Note: by convention, this procedure has only one argument. It provides the
#       parent frame into which you should pack all widgets.
proc    APSMpPARStandardizeInitDialog {frame} {

    # Note: this dialog procedure has access only to those variables explicitly
    #       passed in via the -vars option of APSMpStep. You can access them here
    #       as elements of the following global array (which always has the same
    #       name as the machine procedure itself.
    global APSMpPARStandardize

    APSLabeledEntry .le1 -parent $frame -label "Enter energy" -textVariable \
        APSMpPARStandardize(energy)

    APSLabeledEntry .le2 -parent $frame -label "Enter Restore File Name" \
        -textVariable APSMpPARStandardize(restoreFile)
}
```

This procedure may be added to the PEM's configuration file (pemConfig). When the PEM is invoked, you may select APSMpPARStandardize for execution. If you execute it in Automatic mode, there will be no interaction, although you will see the steps displayed in sequence in a status dialog box. If you execute in Semi-Automatic mode, the status dialog will stop at the "init" step and wait for you to fill out the InitDialog and press OK. All subsequent steps in APSMpPARStandardize will be passed over. If you execute in Manual mode, the InitDialog will also be presented, and the procedure will stop at every APSMpStep, awaiting confirmation.

Now suppose you wish to build a higher level machine procedure called APSMpPARTurnOn. This will in turn call APSMpPARStandardize. There is no problem doing this directly, ex.

```
proc    APSMpPARTurnOn {args} {

    set energy 400
    set restoreFile ""
    APSParseArguments {energy restoreFile}

    if [catch \
        {APSMpPARStandardize -energy $energy -restoreFile $restoreFile} res] {
        puts stderr "error during standardization: $res"
    } else {
        puts "par standardize succeeded" $res"
    }

    # Code to start up PAR RF here

    APSMpReturn ok
}
```

The problem with this, however, is that the code to start up the PAR RF will not begin until the PAR standardization is completed. If starting up the PAR RF were also time consuming, it is clearly best that these two activities occur in parallel. Here is another version using APSMpParallel and APSMpJoin:

```
proc    APSMpPARTurnOn {args} {

    set energy 400
    set restoreFile ""
    APSParseArguments {energy restoreFile}

    set p "APSMpPARStandardize -energy $energy -restoreFile $restoreFile"
    # Note: the next statement causes a separate UNIX process to start which
    #       handles the execution of the procedure. This process will inherit
    #       the host selection and execution mode of the current procedure,
    #       although options to APSMpParallel permit you to override that.
    if [catch {APSMpParallel -procedure $p} result] {
        APSInfo .parOnAlert -name "Warning" -infoMessage \
            "PAR Standardization failed: $result" -modal 1
        APSMpReturn error "unable to run parallel procedure"
    } else {
        set parallelId $result
    }

    # Code to start up PAR RF here

    # This statement will block until the parallel procedure is completed. If you
    #  omit the use of APSMpJoin, it will automatically be done for you prior
    #  to returning from this procedure. Note that return value from parallel
    #  procedure is returned by APSMpJoin.
    catch {APSMpJoin $parallelId} parallelResult

    APSMpReturn ok
}
```

By using APSMpParallel and APSMpJoin, you can construct arbitrary parallelism in your procedures. Multiple parallel procedures may be outstanding at any given time. The invoking procedure can synchronize with them in any order using APSMpJoin. You may wish to use APSMpAbort with the "-onError 1" option to prevent blocking in APSMpJoin's in the event that one of your parallel procedures returns an error.

As stated above, you may omit any use of APSMpJoin. The "system" will automatically perform an APSMpJoin on all outstanding processes prior to returning from your high level procedure. Explicit use of APSMpJoin is best, though, since it permits you to check the return status and value of the parallel procedure.

There are a number of options to APSMpStep, APSMpReturn, and APSMpParallel which are covered in the reference section.

3. Reference

3.1. Utility Procedures

- **APSMpStep** stepName [-promptDialog <procName>] [-vars <varList]
  Defines a step within a procedure. When executed from the PEM, each step name is displayed as it is executed. When a procedure is executed in Automatic mode (or outside of the PEM), steps are ignored. When in Semi-Automatic mode, the PEM stops at the first step encountered in the highest level procedure. If -promptDialog is

specified, <procName> is executed. Execution resumes when the user dismisses the dialog box. The dialog procedure <procName> must have the following structure:

```
proc    <procName> {frame} {
    global <machineProcName>

    # Create widgets and use $frame as parent, i.e.
    APSLabeledEntry .le -parent $frame -label "Enter something" \
        -textVariable <machineProcName>(<varName>)
}
```

In order to permit the promptDialog access to variables in the scope of the machine procedure, you must pass them explicitly with the -vars option. When this is done, a global array variable is created with the same name as the machine procedure. The promptDialog may then access the variables as elements of the array. See the section with examples of machine procedures.
When the PEM execution mode is set to Manual, the PEM will stop at every step. The user must press “Continue” to resume. When a machine procedure returns, the user must press “Done” to complete interaction.

- **APSMpInterface**
  Returns 1 if the current execution environment of your procedure permits the use of graphical widgets. 0 otherwise. This should be used whenever you have code in a machine procedure which, for example, brings up a dialog or alert box.
- **APSMpReturn** code [results]
  This must be called in place of the usual “return”. Use of the tcl “return” will produce errors during execution. The code may be either “ok” or “error”. The results argument may be any arbitrary value you wish to return to the calling procedure.
- <id> **APSMpParallel** -procedure <proc> [-host <hostName>] [-mode <mode>] [-interface 1]
  Creates a separate process containing a tcl interpreter and executes <proc> within the interpreter. The current host and mode are passed to the separate process, unless overridden by explicitly supplying them. If a given <hostName> is supplied, you must have xhost permission set (ie. “xhost + <hostName>”).

  For example, if you are in Manual mode while executing the current procedure, any additional procedure you schedule with APSMpParallel will result in another status dialog appearing. This second status dialog will force you to acknowledge every APSMpStep in the parallel procedure as well. You could override this by supplying “-mode Automatic”, thereby forcing the parallel procedure to execute without requiring any interaction.

  APSMpParallel returns a unique <id> which identifies the parallel process. This may be later used in a (optional) call to APSMpJoin.
- <results> **APSMpJoin** <id>
  Synchronize current execution with parallel procedure <id>. This call will cause your current execution to block until the parallel procedure <id> has completed. Any return value from the parallel procedure will be returned as <results>. If you omit use of APSMpJoin, the PEM will automatically perform a join on all outstanding <id>’s prior to returning from the calling procedure. Note that you should always use a “catch” statement on a join, since an error in the parallel procedure will generate an error when you perform an APSMpJoin.
- **APSMpAbort** <idList> [-onError 1]
  Abort execution of parallel procedure(s) on <idList> (and by extension, any child procedures thereof). If <idList> is keyword “**all**”, all outstanding parallel procedures are aborted. This implements an immediate abort.

The “-onError 1” option requests that all procedures on <idList> be aborted if any one of them returns an error. This is a deferred abort. This prevents subsequent APSMpJoin’s from blocking if an error has occurred.

3.2. Installing PEM

Successful use of the PEM and machine procedures requires a number of things to be in place. Thankfully, these can be done once and forgotten. We will cover these items one by one here. Two OAG software modules must be installed first. One is oag/apps/src/pem, the other is oag/apps/src/tcltklib. Both the documents and source code are available on the WWW under http://www.aps.anl.gov/asd/oag/oagSoftware.html.

3.2.1. Tcl interpreter

We currently depend on the existence of the following minimum versions of Tcl/Tk and their extensions: Tcl7.4, Tk 4.0, dp 3.4b1. In addition, most of the various scripts invoke the interpreter we call “oagwish”. This is our site specific interpreter built against the set of extensions we commonly use. You are free to replace any reference to “oagwish” with your own interpreter. The minimum requirement, however, is that you have dp 3.4b1 (or later versions).

3.2.2. APS Tcl/Tk Library

The APS Tcl/Tk library must be installed first (which includes building a tclIndex). Any and all scripts must set their auto_path to the installation directory. This allows all necessary tcl procedures to be dynamically loaded as needed.

3.2.3. PEM package

This package provides some additional library procedures which are added to the same install directory as the APS Tcl/Tk Library. The “pem” script is here as well. Less obviously, there are two additional important scripts “runMP” and “runMPserv”. Installing the pem package will put these into the bin directory, but you must configure inetd (see UNIX man pages) to access them.

3.2.4. Configuring inetd

Machine procedures may be executed on any host machine on the subnet. In order to permit this, I chose to use inetd as the invoking mechanism. For every host machine you intend to use, you must add the following line to /etc/inetd.conf, and restart inetd:

```
runMP   stream  tcp    nowait  oag    /usr/local/bin/dptcl    dptcl -f /usr/
local/oag/apps/bin/sun4/runMPserv
```

Replace “oag” with whatever account you want, and also the paths may vary for your site.

Basically, the PEM invokes a machine procedure by contacting inetd. Inetd then in turn fires up the “runMPserv” script which sets up an appropriate shell environment. You will probably want to customize runMPserv to set various environment variables, as the inetd environment is extremely vanilla (ie. no cshrc is sourced). This environment you set up is what the interpreter running your machine procedure will have.

The runMPserv script then executes the runMP script. The runMP script takes care of setting up any user interface, and contacting the PEM in order to communicate runtime status.