TclOK User Guide

Version 2025.12

DashThru

Copyright Notice

Contents

1

1 TclOK Introduction

TclOK is a Tcl script check tool developed by DashThru Technology, featuring a novel "static + dynamic" checking approach. It helps users quickly scan for basic syntax errors and other logical errors that may cause script execution to terminate, before running the Tcl script.

As a simple and flexible interpreted language, Tcl (Tool Command Language) is widely used in various chip design EDA (Electronic Design Automation) tools. Due to the lack of a compilation process, errors in Tcl scripts are only discovered during the execution of the script within the EDA tool. In practical chip design projects, Tcl scripts can contain thousands of lines of code and may be nested in multiple hierarchies, making it difficult to detect hidden syntax errors. When these Tcl scripts are used to perform tasks such as chip synthesis or place and route, errors in the Tcl code often cause the EDA tool to run for an extended period before unexpectedly terminating.

Traditional Tcl checking tools rely solely on static analysis, such as checking for spelling mistakes or missing semicolons and parentheses, which can only identify basic syntax errors. These tools are ineffective in detecting advanced logical errors. Therefore, debugging Tcl scripts is challenging for engineers, who often need to repeatedly execute the scripts in the EDA tool for iterative debugging, consuming valuable project development time.

TclOK uses a "static + dynamic" checking method. By virtually executing the Tcl script, it can detect most errors within seconds and provide clear descriptions and locations of the issues, allowing users to quickly resolve Tcl script problems.

1-1 TclOK Feature Overview

To understand the specific functionality of TclOK for Tcl script check, please refer to Section 3: TclOK Features. TclOK has the following features:

- **Static Check:** Performs structural validation of Tcl commands, including basic syntax, argument types, and quantity checks.
- **Dynamic Check:** Uses virtual execution mechanism to handle EDA-tool specific commands which is unknown for TclOK.
- **Checking Modes:** Offers both default and verbose checking modes.
- **SDC/UPF Support:** Native support for SDC (Standard Design Constraints) and UPF (IEEE-1801) formats.
- **Initialization Script:** Allows specification of TclOK initialization scripts at startup, supporting both Tcl and Python formats.
- **Support for Auto Run:** Supports automatic execution of third-party programs, such as EDA tools, upon successful validation.

1-2 TclOK Installation and Deployment

- **Recommended Operating System Versions: RHEL/CentOS 6.5-7.9**
  Experimental support for RHEL/Rocky/Alma 8.x

To check the system version on RHEL and CentOS systems, view the `system-release` file

```
% cat /etc/system-release
CentOS Linux release 7.9.2009 (Core)
```

- **Installing TclOK**

The DashRTL package includes TclOK. Simply extract the compressed package to the installation directory.

```
% tar xJvf DashRTL_v2024.12.tar.xz
```

- **Setting System Variables**

System variables can be added to the shell initialization file (e.g., `.cshrc`). In the following example, replace `<installation_path>` with the actual installation path. After setting this, you can directly use the `tclok` executable from the command line.

`DASHTHRU_LICENSE_SERVER`: Set this for the license server. Please refer to the **DashLM User Guide** for details on how to obtain a license and start the license service. `<port>`

is the license service port, with a default of 28000. `<hostname>` refers to the machine name or IP address where the DashLM license service is running.

```
%  setenv DASHTHRU_LICENSE_SERVER <port>@<hostname>:…
%  set path = ($path <installation_path>/DashRTL_v2024.12/bin)
```

The example for csh is as follows:

```
%  setenv DASHTHRU_LICENSE_SERVER 28000@lic_server1:28000@lic_server2
%  set path = ($path /edatool/DashRTL_v2024.12/bin)
```

- **Starting TclOK**

```
%  which tclok
/edatool/DashRTL_v2024.12/bin/tclok
%  tclok test.tcl -v
TclOK(TM) Script Checker
Copyright(c) 2024, DashThru Technology, Ltd. All rights reserved.

Version:    v2024.12-Alpha, build 2024/12/20
Date:       2024/12/25 23:02:22
Host:       EPYC / 64 Threads
Launch:     tclok test.tcl -v
Feedback:   support@dashthru.com

Info: checked out license 'FusionShell-Lint'. (LIC-CO)

****************** TCL CHECK SUMMARY ******************
 Total errors                : 0
 Total unknown commands      : 0
 Possible unknown commands   : 0
 Hierarchical summary        : test.tcl (errors:0, unknown_cmds:0)

OK
```

2

2 TclOK Check Mechanism

TclOK employs an innovative "static + dynamic" checking approach.

In terms of static checking, TclOK performs formal checks on commands, including basic syntax validation such as spelling, matching parentheses and quotation marks, variable assignment and references, and command argument types.

For dynamic checking, to ensure high-speed checking efficiency, TclOK virtually executes commands. This allows TclOK to check the validity of a command without actually executing it, and to propagate the command's return value. TclOK also introduces a new concept of returning an "unknown" value: when an unknown command is virtually executed, its return value is set to "unknown" and is propagated through the entire script.

This chapter will provide a detailed overview of TclOK's checking principles, covering the following topics:

- Overview of TclOK Check Process
- Handling Unknown Commands
- User-Defined Commands

2-1 Overview of TclOK Check Process

The TclOK checking process consists of two steps: **formal command validation** and **virtual command execution**.

Formal Validation focuses on analyzing basic syntax, such as checking for spelling errors, matching parentheses and quotation marks, variable assignment and references, and command argument types.

Virtual Execution handles three possible scenarios: normal execution, normal skipping and unknown command handling.

- **Formal Command Validation**

Bracket and Quotation Matching: This check ensures that all parentheses and quotation marks are properly matched. For example, the following command has a missing quote and bracket:

```
puts "haha
puts {haha
```

Incorrect Variable References: This check identifies references to non-existent variables. In the following example, the variable `b` does not exist:

```
set a 0
puts $b
```

Command Argument Count: TclOK verifies the correct number of arguments for commands. For instance, the `llength` command is incorrectly used with two arguments:

```
llength {a b c} {a b}
```

Command Argument Value Validity: This check validates the correctness of command argument values. For example, the second argument `haha` in the following command is not a valid channel:

```
puts haha haha
```

Command Argument Type: TclOK checks the argument types for commands to ensure they match the expected types. In the following example, the second argument should be an integer:

```
lindex {a b c} a
```

- Virtual Command Execution

The virtual execution of commands in TclOK is divided into three scenarios: **Normal Execution**, **Normal Skipping**, and **Unknown Command Handling**.

Normal Execution: This includes the execution of native Tcl commands, user-defined proc commands, as well as SDC and UPF commands. These commands are executed as intended by the Tcl interpreter.

Normal Skipping: Commands that involve disk write operations or those executed via the exec command (which runs external processes) are skipped during virtual execution. These commands are not executed, but TclOK will handle them by bypassing their execution without triggering an error.

Unknown Command Handling: If a command is unknown, it is not treated as an error like in the standard Tcl Shell. Instead, TclOK assigns an "unknown" return value to the command, which is then propagated. This is a unique handling mechanism of TclOK. For more details, please refer to Section 2-2: Handling Unknown Commands.

- Virtual Execution of Tcl Commands and Handling of External Commands

Virtual Execution of Native Tcl Commands:
Native Tcl commands, such as `list`, are virtually executed and return valid values as expected. For example, in the following code, the arguments for the list command on lines 2 and 3 return valid results:

```
set var [lindex {a b c} 1]
list $var
list [lindex {a b c} 1]
```

Virtual Execution of User-Defined Procs:
User-defined `proc` commands also return valid values during virtual execution. For example, the user-defined `p` procedure below returns a value as expected, allowing the arguments on lines 3 and 4 to produce valid results:

```
proc p {} {return [lindex {a b c} 1]}
set var [p]
list $var
list [p]
```

Skipping Commands Involving Disk Operations or External Exec Commands:
Commands that involve disk write operations (e.g., file deletion) or external exec

commands are skipped during virtual execution. In the example below, the three commands are not executed:

```
file delete test.tcl
exec rm test.tcl
exec python3 ext.py
```

If a user needs to execute external `exec` commands as an exception, the `-enable_ext_exec` option can be used. For detailed usage, please refer to Appendix A: TclOK Options List.

2-2 Handling Unknown Commands

- **Identification of Unknown Commands**

In addition to native Tcl commands and user-defined `proc` commands, TclOK also includes built-in checks for SDC and UPF commands. Any other commands are identified as unknown.

It is common for Tcl scripts, particularly in the context of EDA tools, to contain a large number of unknown commands, which are typically proprietary commands specific to the EDA tools. These commands are not errors in the Tcl script because they are designed to work within the EDA tools. TclOK aggregates information about these unknown commands and prints them out, allowing users to review and manage them.

Unknown commands recognized by TclOK are displayed in the **summary report** (when using the `-verbose` flag). In the example below, the `test.tcl` script file contains two types of unknown commands: **unknown commands** and **possible unknown commands**, which are listed separately in the summary.

Many EDA tools Tcl shells allow direct invocation of Linux commands, such as the common `ls`, `rm`, `which`, `hostname`, etc. However, in some EDA tools, these commands are treated as built-in commands, making it difficult to determine whether they are unknown commands. Therefore, they are classified as **possible unknown commands**. Other true unknown commands are classified as **unknown commands**.

test.tcl

```
set var [set_module_top top]
set_parameter -top true
set hn [hostname]
hostname
```

```
%  tclok test.tcl -verbose
……………
****************** TCL CHECK SUMMARY ******************
  Total errors               : 0
  Total unknown commands     : 2 ->'set_module_top', 'set_parameter'
  Possible unknown commands  : 1 ->'hostname'
  Hierarchical summary       : test.tcl (errors:0, unknown_cmds:2
  ->'set_module_top', 'set_parameter')
```

- **Virtual Execution of Unknown Commands**

Once unknown commands are recognized, they cannot be executed normally. TclOK virtually executes them and returns an "unknown" value. In the following example, an unknown command `unknown_cmd` is executed, and its return value is "unknown," which causes the `list` command to also receive an "unknown" value as an argument:

```
set var [unknown_cmd {a b c}]
list $var
list [unknown_cmd {a b c}]
```

The "unknown" value returned from the virtual execution of unknown commands propagates through the command arguments. In the example below, the "unknown" value returned by the `unknown_cmd` command is passed to the `lindex` command’ s argument, causing the return value of `lindex` to also be "unknown," and the final `list` command will also receive "unknown" as its argument:

```
set var [lindex [unknown_cmd {a b c}] 1]
list $var
list [lindex [unknown_cmd {a b c}] 1]
```

If TclOK is used with the default settings without handling unknown commands, the "unknown" values propagated from the virtual execution of unknown commands may degrade the validity of the entire Tcl script check. This is similar to the propagation of "X-state" in digital chip simulations, which could lead to unreliable simulation results for the entire design.

Therefore, we recommend that users register these unknown commands as known commands in the initialization file. This not only eliminates the propagation of "unknown" values but also allows for detailed format checks on the command options. For information on how to define user-customized commands, please refer to Section 2-3: User-Defined Commands.

2-3 User-Defined Commands

User Tcl scripts often contain a large number of unknown commands, typically specific to EDA tools. As mentioned in **Section 2-2**, when TclOK virtually executes an unknown command, it may propagate the "unknown" return value extensively, which can reduce the overall effectiveness of the Tcl script check.

It is recommended that users register these unknown commands as known commands in the initialization file. The benefits of user-defined commands are as follows:

1. By setting the return value of the command in the proc definition to a fixed value that mimics the behavior of the EDA tool, the propagation of "unknown" values is prevented, which improves the accuracy of Tcl script checking.

2. EDA tool commands typically have a large number of options. These can be formally defined within the proc, allowing for detailed formatting of the command and enabling the formal checking of argument types.

- **Example of Customizing Commands Using Tcl Initialization Script**

To define custom commands using a Tcl initialization script, the commands `parse_proc_args` and `define_proc_constraints` are required. For detailed usage, please refer to **Section 2-3: Tcl Language Extensions: Proc Command Extensions** in the **FusionShell User Guide**.

In the example below, the script to be checked, `test.tcl`, contains two unknown commands: `set_parameter` and `hostname`. We can prepare an initialization script, `init.tcl`, to register these commands as known commands.

In the initialization script, we define the `set_parameter` command and use the `define_proc_constraints` command to define the format of its options, such as the option name, type, default value, and whether it is optional. We then use the `parse_proc_args` command to pass the corresponding option values to the variables associated with each option. Finally, we output the return value of the `set_parameter` command, `$cellname`. As a result, in the first line of the command in `test.tcl`, the `set_parameter` command will not return an "unknown" value, preventing the propagation of "unknown" values to the `result` variable.

Additionally, we define the `hostname` command, which returns a fixed value, `machine001`. As a result, the summary in the check report will show that the number of "unknown commands" is zero.

test.tcl

```
set result [set_parameter -cellname top -value 123]
set_parameter -iscell true
set hn [hostname]
```

init.tcl

```
proc set_parameter {args} {
      parse_proc_args -to_vars
      return $cellname
}

define_proc_constraints set_parameter \
      -switch_arg "name=cellname type=string optional=true default=" \
      -switch_arg "name=value type=int default=1" \
      -switch_arg "name=iscell type=bool"

proc hostname {} {
      return machine001
}
```

```
%  tclok test.tcl -init init.tcl -verbose
...............
****************** TCL CHECK SUMMARY ******************
  Total errors               : 0
  Total unknown commands     : 0
  Possible unknown commands  : 0
  Hierarchical summary       : test.tcl (errors:0, unknown_cmds:0)
```

- **Example of Command Argument Type Checking Using Tcl Initialization Script**

If we use the same initialization file, `init.tcl`, to define the user's commands, we can specify argument types for the `set_parameter` command. For instance, we define that the `-cellname` option only accepts data of type string, the `-value` option only accepts int type data, and the `-iscell` option only accepts `boolean` type data.

In the following Tcl script to be checked, the `set_parameter` command incorrectly specifies the data types for the `-value` and `-iscell` options. The TclOK check report clearly highlights these two errors: 1.2 is not an integer type, and top is not a boolean type.

test.tcl

```
set result [set_parameter -cellname top -value 1.2]
set_parameter -iscell top
set hn [hostname]
```

```
%  tclok test.tcl -init init.tcl -verbose
……………
checking: set_parameter -cellname top -value 1.2
Error: wrong # args: expect integer value for '-value' of procedure
  'set_parameter' but get '1.2'
 Line: 1
 File: test.tcl

checking: set_parameter -iscell top
Error: wrong # args: expect boolean value for '-iscell' of procedure
  'set_parameter' but get 'top'
 Line: 1
 File: test.tcl

****************** TCL CHECK SUMMARY ******************
  Total errors               : 0
  Total unknown commands     : 0
  Possible unknown commands  : 0
  Hierarchical summary       : test.tcl (errors:0, unknown_cmds:0)
```

- **Example of Customizing Commands Using Python Initialization Script**

TclOK also supports using Python scripts as initialization scripts to define custom commands. In the example below, we use `init.tcl` as the initialization script, but first, we need to switch to Python mode using the `pymode` command. After switching to Python mode, we register the `say_one_or_two_words` command as a known command and define its options and argument types.

```
init.tcl_py

pymode

from tcl import TclProcConstrExt, TclArgConExt, TclArgConStd

tcl._constr.proc_cons['say_one_or_two_words'] = TclProcConstrExt(
    'say_one_or_two_words', 'proc info not needed', [
        TclArgConExt('capitalize', multi=False, type_con=bool),
        TclArgConStd('word', optional=False),
        TclArgConStd('word2', optional=True)
    ])

tcl._procs['say_one_or_two_words'] = lambda arg_dict: ''
```

```
test.tcl

set worda [say_one_or_two_words a]
say_one_or_two_words a b
say_one_or_two_words -capitalize 1 a b
```

```
%  tclok test.tcl -init init.tcl -verbose
...............
<info> switch back to tclmode ...
****************** TCL CHECK SUMMARY ******************
  Total errors               : 0
  Total unknown commands     : 0
  Possible unknown commands  : 0
  Hierarchical summary       : test.tcl (errors:0, unknown_cmds:0)
```

3

3 TclOK Features

TclOK checks are divided into **Default Check Mode** and **Verbose Check Mode**. The Default Check Mode only prints essential information such as Tcl errors, making it easy for users to review. The Verbose Check Mode includes additional information in the summary, providing users with an overview of the Tcl script.

TclOK supports the execution of initialization scripts before checking the target Tcl script. These scripts allow users to set necessary environment variables and define unknown commands. Initialization scripts can be written in either Tcl or Python, and can be specified via the `-init` option to provide a script file or via the `-x` option to specify script commands.

TclOK also supports automatically running third-party programs (such as EDA tools) upon successful validation. This feature allows users to check the correctness of the Tcl script before launching the third-party tool each time.

This chapter provides a detailed overview of TclOK's features, including the following topics:

3-1 Default Check Mode

When TclOK is used without the `-verbose` option, it enters **Default Check Mode** and uses the `-log` option to specify the output log file. In Default Check Mode, only the most basic Tcl check information is printed, such as Tcl errors and "unknown command" warnings. The usage is as follows:

```
% tclok test.tcl -log tclcheck.log
```

In the example for Default Check Mode below, the information printed on the screen and in the log file differs slightly. The log file includes additional "checking: ..." lines, which provide information about the commands being checked, while the screen output only shows concise error messages for easier user review.

Whether printed on the screen or in the log file, if there are any errors in the Tcl script check, the result will display `FAILED` at the end. If no errors are found, `OK` will be displayed at the end.

test.tcl

```
set a 0
puts $b
set result [set_parameter -cellname top -value 1.2]
set_parameter -iscell top
```

Screen

```
%   tclok test.tcl -log tclcheck.log
checking: set a 0
Error: can't read "b": no such variable
  Line: 2
  File: test.tcl

Warning: script contains an unknown command 'set_parameter' of which check has been skipped.

FAILED
```

tclcheck.log

```
checking: set a 0

Error: can't read "b": no such variable
  Line: 2
  File: test1.tcl

checking: set_parameter -cellname top -value 1.2
checking: set result ...
checking: set_parameter -iscell top
Warning: script contains an unknown command 'set_parameter' of which
check has been skipped.

FAILED
```

3-2 Verbose Check Mode

When TclOK is used with the `-verbose` option, it enters **Verbose Check Mode** and uses the `-log` option to specify the log file for output. Compared to Default Check Mode, Verbose Check Mode adds additional information about unknown commands and includes a `TCL CHECK SUMMARY` section. The usage is as follows:

```
% tclok test.tcl -log tclcheck.log -verbose
```

In the example for Verbose Check Mode below, the `TCL CHECK SUMMARY` information is described as follows:

Total errors: Displays the total number of Tcl script errors.

Total unknown commands: Shows the number and names of unknown commands.
-- **affecting branch**: Lists the number and names of unknown commands that affect conditional statements.
-- **affecting loop**: Lists the number and names of unknown commands that affect loop statements.

Possible unknown commands: Shows the number and names of possible unknown commands.

Hierarchical summary: Provides detailed information about nested script files (those using `source` to include other scripts).

This level of detail in Verbose Check Mode provides users with a comprehensive view of the Tcl script's execution and potential issues.

test.tcl

```
set a 0
puts $b
set result [set_parameter -cellname top -value 1.2]
set_parameter -iscell top
```

Screen

```
%    tclok test.tcl -log tclcheck.log -verbose
checking: set a 0
Error: can't read "b": no such variable
  Line: 2
  File: test1.tcl

<info> line 2: calling unknown command 'set_parameter', returning result will be
unknown.
checking: set result ...
<info> line 2: #2 argument of command 'set' is
'UNKNOWN_returned_by_[set_parameter ...]', returning result will be unknown
****************** TCL CHECK SUMMARY ******************
    Total errors                  : 1
    Total unknown commands        : 1 ->'set_parameter'
    Possible unknown commands     : 0
    Hierarchical summary       : test.tcl (errors:1, unknown_cmds:1 ->'set_parameter')

FAILED
```

tclcheck.log

```
checking: set a 0
Error: can't read "b": no such variable
  Line: 2
  File: test1.tcl

checking: set_parameter -cellname top -value 1.2
<info> line 2: calling unknown command 'set_parameter', returning result will be
unknown.
checking: set result ...
<info> line 2: #2 argument of command 'set' is
'UNKNOWN_returned_by_[set_parameter ...]', returning result will be unknown
checking: set_parameter -iscell top
****************** TCL CHECK SUMMARY ******************
    Total errors                  : 1
    Total unknown commands        : 1 ->'set_parameter'
    Possible unknown commands     : 0
    Hierarchical summary          : test.tcl (errors:1, unknown_cmds:1
                                    ->'set_parameter')

FAILED
```

- **Log Information in Verbose Check Mode**

In Verbose Check Mode, the log output provides more detailed information than the screen output. Below is an example of the additional content included in the log:

1. **checking (at the start of a line)**: Provides a detailed breakdown of the command execution within loops, including `while`, `for`, and `foreach` loops. This helps trace the execution flow and identify potential issues within loop structures.

2. **in_proc (at the start of a line)**: Provides detailed information about the execution of commands within user-defined `proc` commands. This allows users to see how commands inside custom procedures are being processed and whether any issues arise there.

3. **(expr) (at the start of a line)**: Shows detailed information about the evaluation of expressions. This includes the evaluation of conditions, logical expressions, and any other evaluated code, helping users to track the logic flow and diagnose any issues related to expression evaluations.

tclcheck.log

```
checking: set i 0
checking: proc add i {return [incr $i]}
checking: while {$i<2} ...
( expr ): { $i<2 }
( expr ): <internal> return 1
checking: while...add 0
  in_proc: while...incr 0
  in_proc: while...return 1
checking: while...set str {? 1}
checking: while...puts {? 1}
checking: while...incr i
```

3-3 Strict Check Mode

TclOK performs two types of checks when verifying Tcl scripts: **default mode** and **strict mode**. In default mode, TclOK does not raise an error when it encounters an unknown command. Instead, it treats the unknown command using a virtual execution mechanism. For more details, please refer to Section 2-2 Handling Unknown Commands.

However, if all third-party tool commands used in the script have been defined and the user considers any other unknown commands to be invalid, they may want these commands to be reported as errors. In this case, the user can enable **strict mode** using the `-strict` option. In strict mode, any unknown command encountered will result in an error, and the final check result will be marked as `FAILED`. Additionally, the third-party programs specified by the `-autorun` option will not be launched.

```
top.tcl
.........
# Correct command name should be get_pins/set_level_shifter

set subin [getpins sub/in]
set_level_shifters -domain pdsub -applies_to inputs -location self sub/in
```

In the Tcl script `top.tcl` that is to be checked, there are two commands, `get_pins` and `set_level_shifter`, which are mistakenly written as `getpins` and `set_level_shifters`. Below are the results for both check modes.

- **Default Check Mode**

In default check mode, no error is raised. The two incorrect commands are identified as unknown commands, and the check result is `OK`, as shown below:

```
%  tclok top.tcl -verbose
...............
****************** TCL CHECK SUMMARY ******************
  Total errors                 : 0
  Total unknown commands       : 2 ->'getpins', 'set_level_shifters'
  Possible unknown commands    : 0
  Hierarchical summary         :top.tcl (errors:0,
  unknown_cmds:2 ->'getpins', 'set_level_shifters')

OK
```

- **Strict Check Mode**

When the `-strict` option is used, the two incorrect commands will not be treated as unknown commands and will instead trigger an error. The check result is `FAILED`, as shown below:

```
%  tclok top.tcl -verbose -strict
……………
Error: invalid command name "getpins"
 Line: 3
 File: top.tcl

Error: invalid command name "set_level_shifters"
 Line: 4
 File: top.tcl

****************** TCL CHECK SUMMARY ******************
  Total errors               : 2
  Total unknown commands     : 0
  Possible unknown commands  : 0
  Hierarchical summary       : top.tcl (errors:2, unknown_cmds:0)

FAILED
```

3-4 Initialization Script Execution at Startup

TclOK supports the execution of user-specified initialization scripts before checking the target Tcl script. Common scenarios for executing initialization scripts include:

1. Registering unknown commands as known commands to avoid the impact of "unknown command" and the propagation of "unknown" return values. See Section 2-3: User-Defined Commands.
2. Setting up general environment variables, such as tool-specific variables for third-party EDA tools.

Initialization scripts can be written in either Tcl or Python. The script file can be specified using the `-init` option, or individual script commands can be provided using the `-x` option.

Usage of Tcl Initialization Script:

```
% tclok test.tcl -log tclcheck.log -verbose -init init.tcl
% tclok test.tcl -log tclcheck.log -verbose -x "set design top;
set width 16;"
```

Usage of Python Initialization Script:
Since Tcl variables are always treated as strings, when setting variables in the Python initialization script, they must also be of string type:

```
% tclok test.tcl -log tclcheck.log -verbose -init init.tcl
% tclok test.tcl -log tclcheck.log -verbose -x "pymode;
design='top'; width='16';"
```

These options allow users to initialize their environment or register commands before the Tcl script check, enabling a smoother and more customized checking process.

- **Example of TclOK Executing Initialization Script**

In the following example, the Tcl script `test.tcl` to be checked uses unknown variables `$design` and `$width`, as well as the unknown command `set_parameter`. These are defined through the initialization script file `init.tcl`, which supports both Tcl and Python script modes.

```
set version 1.0                                              test.tcl
puts $width
set result [set_parameter -cellname $design -value 1.2]
set_parameter -iscell top
```

The example of a Tcl initialization script file is as follows:

```
% tclok test.tcl -log tclcheck.log -verbose -init init.tcl
```

```
set design top                                               init.tcl
set width 16
proc set_parameter {args} {return 1}
```

The example of a Python initialization script file is as follows:

```
% tclok test.tcl -log tclcheck.log -verbose -init init.tcl
```

```
pymode                                                       init.tcl
design='top'
width='16'
tcl._constr.proc_cons['set_parameter'] = ……
tcl._procs['set_parameter'] = lambda arg_dict: '1'
```

- **Example of TclOK Executing Initialization Script**

In the example below, the Tcl script `test.tcl` to be checked uses unknown variables `$design` and `$width`, as well as an unknown command `set_parameter`. The initialization script is executed directly using the `-x` option to define these variables and commands. TclOK supports both Tcl and Python initialization scripts.

```
test.tcl
set version 1.0
puts $width
set result [set_parameter -cellname $design -value 1.2]
set_parameter -iscell top
```

The example of executing the Tcl initialization script using `-x` is as follows:

```
%  tclok test.tcl -log tclcheck.log -verbose -x "set design top;
set width 16;"
```

The example of executing the Python initialization script using `-x` is as follows:

```
%  tclok test.tcl -log tclcheck.log -verbose -x "pymode;
design='top'; width='16';"
```

3-5 Print Variable Values at Specific Line Numbers

TclOK supports printing the value of a specified variable at a specific line in the file, which helps users analyze the execution process and variable state of the script. By using the `-print_var_at_line` option to specify the line number and variable name, TclOK will print the value of the variable at that particular line in the script.

test.tcl

```
1   set FLOW floorplan
2   set DESIGN_TOP sub
3
4   # Design Settings
5   if {$FLOW == "floorplan"} {
6       if {$DESIGN_TOP == "top"} {
7           set result [set_parameter -cellname top -value 1.2]
8           set_parameter -iscell true
9       }
10  } else {
11      set result [set_parameter -cellname top -value 1.2]
12      set_parameter -iscell false
13  }
```

In the above Tcl script example, the user needs to determine whether the `if` statement on line 5 of `test.tcl` was executed. To do this, the user may want to know the values of certain variables at that point. The following methods can be used to print these variable values.

If there is only one script file, the file name can be omitted:

```
% tclok test.tcl -verbose -print_var_at_line "5 FLOW DESIGN_TOP
  RUN_TYPE"
```

A wildcard (`*`) can be used to match file names, such as in the following usage, which prints the variable values from line 5 of files starting with `test`:

```
% tclok test.tcl -verbose -print_var_at_line "test*:5 FLOW
  DESIGN_TOP RUN_TYPE"
```

When there are multiple script files (e.g., due to `source` nesting), it is recommended to specify the full path of the file:

```
% tclok test.tcl -verbose -print_var_at_line "test.tcl:5 FLOW
  DESIGN_TOP RUN_TYPE"
```

Using the above methods, TclOK prints the following variable values to the screen and log:

```
<hook> line 5: value of $FLOW is "floorplan"
<hook> line 5: value of $DESIGN_TOP is "sub"
<hook> line 5: variable $RUN_TYPE does not exist
```

It is important to note that if the specified line does not contain any commands (such as line 3 in the above example), TclOK will search downward for the next line containing a command to report on, which in this case is line 5.

If you want to print variable information at multiple line numbers, you can repeat the `-print_var_at_line` option, as shown below:

```
%  tclok test.tcl -verbose
-print_var_at_line "test.tcl:2 FLOW DESIGN_TOP RUN_TYPE"
-print_var_at_line "test.tcl:5 FLOW DESIGN_TOP RUN_TYPE"
```

In this case, TclOK will print the variable values for both line 2 and line 5 as follows:

```
<hook> line 2: value of $FLOW is "floorplan"
<hook> line 2: variable $DESIGN_TOP does not exist
<hook> line 2: variable $RUN_TYPE does not exist
<hook> line 5: value of $FLOW is "floorplan"
<hook> line 5: value of $DESIGN_TOP is "sub"
<hook> line 5: variable $RUN_TYPE does not exist
```

3-6 UPF Check Mode Control

TclOK provides two modes for UPF (Unified Power Format) checks:

1. **Normal Mode:** This mode allows UPF files to contain non-UPF standard commands, such as `get_cells`, `get_pins`, and other Tcl commands that may not strictly conform to UPF standards.
2. **Strict Mode:** In this mode, non-UPF standard commands are not allowed. Only UPF-compliant commands are permitted, ensuring stricter validation and adherence to UPF specifications.

These two modes give users flexibility in how UPF files are checked, allowing either more lenient checks or a more stringent, standards-compliant validation process.

test.tcl
```
………
load_upf top.upf
………
```

top.upf
```
………
set subin [get_pins sub/in]
set_level_shifter -domain pdsub -applies_to inputs -location self $subin
set_level_shifter -domain pdsub -applies_to outputs -location parent [get_cells
sub/out]
```

By default, TclOK uses **Normal Mode** for UPF checks, which allows non-UPF standard commands (such as `get_cells`, `get_pins`, etc.) in UPF files. In the example of the `top.upf` file, no errors will be raised in Normal Mode. However, if you need to use **Strict UPF Mode**, the `-strict_upf` option must be specified.

Strict UPF Mode affects the UPF file check in two ways:

1. **Direct Check on .upf Files:** This checks the file with the `.upf` extension directly.
2. **Check on Files Specified by load_upf:** This checks UPF files specified using the `load_upf` command in the Tcl script.

Usage examples for enabling Strict UPF Mode are as follows:

```
% tclok top.upf -verbose -strict_upf
% tclok test.tcl -verbose -strict_upf
```

When Strict UPF Mode is enabled, both methods will report the following errors:

```
Error: command 'get_pins' is not allowed inside UPF file
 Line: 30
 File: top.upf

Error: can't read "subin": no such variable
 Line: 31
 File: top.upf

Error: command 'get_cells' is not allowed inside UPF file
 Line: 32
 File: top.upf
```

These errors are raised because Strict UPF Mode only allows UPF-compliant commands, and non-UPF standard commands (like `get_cells` and `get_pins`) are flagged as errors.

3-7 Behavior of Unknown Commands in Conditional and Loop Statements

- **Behavior of Unknown Commands in Conditional Statements**

When an unknown command appears in the condition expression of an `if`, `for`, or `while` statement, resulting in an "unknown" value for the expression, the default behavior of TclOK is to skip the entire `if`, `for`, or `while` statement. This occurs because TclOK cannot determine which branch of the `if` statement should be executed, nor can it determine whether the `for`/`while` loop should exit.

The best solution in this case is to register the unknown command as a known command in the initialization file (`-init init.tcl`) and assign it an actual return value. This allows TclOK to evaluate the expression and execute the `if`, `for`, or `while` statement based on the actual result. For detailed instructions on setting this up, refer to Section 2-3: User-Defined Commands.

If the user does not wish to define custom commands, the behavior of TclOK can be globally configured using the `-conditional_test_unknown_action` option. This option provides three possible settings:

skip: When the expression evaluates to "unknown," the entire `if`, `for`, or `while` statement is skipped, which is the default behavior of TclOK.

as_true: When the expression evaluates to "unknown," the expression is treated as `true`, and the `if`, `for`, or `while` statement is executed as if the condition were true.

as_false: When the expression evaluates to "unknown," the expression is treated as `false`, and the `if`, `for`, or `while` statement is executed as if the condition were false.

These options allow users to control how TclOK handles unknown commands in conditional expressions, providing more flexibility in script validation.

```
test.tcl
if {[unknown_cmd_1] == 0} {
    set a 0
} elseif {[unknown_cmd_2] == 1} {
    set a 1
} else {
    set a 2
}}

while {[unknown_cmd_3] < 10} {
    set a 0
    puts $a
}
```

As shown in the example above, when the condition expressions in if or while commands contain three unknown commands (`unknown_cmd_1`, `unknown_cmd_2`, and `unknown_cmd_3`), the expression evaluates to "unknown."

The following tool behaviors occur based on the three different settings for `-conditional_test_unknown_action`:

1. Using skip: the tool skips the entire `if` or `while` statement, which is consistent with the default behavior when `-conditional_test_unknown_action` is not used.

```
% tclok test.tcl -verbose -conditional_test_unknown_action skip
```

2. Using as_true: in this case, the first `if` expression is treated as `true`, so `set a 0` is executed. The `while` expression is also treated as `true`, but the loop executes only once before it terminates.

```
% tclok test.tcl -verbose -conditional_test_unknown_action as_true
```

3. Using as_false: in this case, both the first and second `if` expressions are treated as `false`, so `set a 2` is executed. The `while` expression is also treated as `false`, so the loop is immediately exited without executing.

```
% tclok test.tcl -verbose -conditional_test_unknown_action as_false
```

- **Behavior of Unknown Commands in Loop Statements**

When an unknown command is executed within a `for`, `foreach`, or `while` loop, TclOK's default behavior is to immediately break out of the loop. This occurs because TclOK cannot determine whether the unknown command is a real error or an undeclared built-in command from an EDA tool.

The best solution in this case is to register the unknown command as a known command in the initialization file (`-init init.tcl`) and assign it an actual return value. This allows TclOK to evaluate the expression and execute the `for`, `foreach`, or `while` loop as expected. For detailed instructions on setting this up, refer to Section 2-3: User-Defined Commands.

If the user does not want to define custom commands, the behavior of TclOK can be globally configured using the `-loop_body_unknown_action` option. The default setting is `delay_break` if the option is not specified. This option provides three possible settings:

delay_break: When an unknown command is encountered, TclOK waits until the current iteration of the loop is completed, and then it breaks out of the loop after executing all the commands in that iteration.

break: When an unknown command is encountered, the loop immediately exits, which is the same behavior as the `break` command.

go_on: When an unknown command is encountered, TclOK ignores the command and continues to execute the loop until the loop’ s exit condition is met.

These settings allow users to control how TclOK handles unknown commands within loops, providing flexibility in script validation and execution.

```
test.tcl

foreach i {0 1 2 3} {
    set_parameter -cellname top -value 1.2
    set var_a $i
}

for {set i 0} {$i < 4} {incr i} {
    set_parameter -cellname top -value 1.2
    set var_b $i
}
```

As shown in the example above, when a `for`, `foreach`, or `while` loop executes the unknown command `set_parameter`, the following tool behaviors occur based on the three different settings for `-loop_body_unknown_action`:

1. Using break: When the tool encounters the unknown command, it immediately exits the `for`/`foreach` loop. As a result, the commands `set var_a` and `set var_b` are not executed, and after execution, these two variables do not exist. This behavior is the same as when `-loop_body_unknown_action` is not used.

```
% tclok test.tcl -verbose -loop_body_unknown_action break
```

2. Using delay_break: When the tool encounters the unknown command, it completes the current iteration of the loop (where `$i` equals 0) and then exits the loop. After execution, the values of `var_a` and `var_b` will be 0, as the loop runs for `$i=0` before exiting.

```
% tclok test.tcl -verbose -loop_body_unknown_action delay_break
```

3. Using go_on: When the tool encounters the unknown command, it continues to execute the entire `for`/`foreach` loop until the loop condition is met. The loop will run for values of `$i` from 0 to 3. After execution, the values of `var_a` and `var_b` will both be 3, as the loop executes through all iterations.

```
% tclok test.tcl -verbose -loop_body_unknown_action go_on
```

3-8 Automatically Running Third-Party Programs After Successful Validation

TclOK supports automatically running third-party programs, such as EDA tools, after a successful script check. This allows users to ensure the correctness of the Tcl script used by the tool before launching the third-party program.

- If the check passes (i.e., TclOK detects no errors in the Tcl script), it guarantees that the third-party tool can run the script without issues.
- If the check fails (i.e., TclOK detects errors in the Tcl script), the third-party tool will not be started, and the user must fix the Tcl script errors before rerunning the check.

To use this feature, the `-autorun` option must be specified along with the third-party program to be executed. In the example below, the Tcl check passes without errors, and then the `run_tool -f test.tcl` program is automatically executed.

```
%  tclok test.tcl -log tclcheck.log -verbose -autorun run_tool
-f test.tcl
..................
****************** TCL CHECK SUMMARY ******************
  Total errors               : 0
  Total unknown commands     : 1 ->'set_parameter'
  Possible unknown commands  : 1 ->'hostname'
  Hierarchical summary       : test.tcl (errors:0, unknown_cmds:1
  ->'set_parameter')

OK, autorun is launching ' run_tool ...'
..................
```

If there are errors detected during the Tcl check, the `run_tool -f test.tcl` program will not be executed. This ensures that the `test.tcl` script is correct before it is passed to the third-party tool. As shown in the example below, if any errors are found in `test.tcl`, the tool will not run.

```
%  tclok test.tcl -log tclcheck.log -verbose -autorun run_tool
-f test.tcl
Error: can't read "value": no such variable
 Line: 1
 File: test.tcl
..................
```

```
****************** TCL CHECK SUMMARY ******************
 Total errors              : 1
 Total unknown commands    : 1 ->'set_parameter'
 Possible unknown commands : 1 ->'hostname'
 Hierarchical summary      : test.tcl (errors:1, unknown_cmds:1
 ->'set_parameter')

FAILED
```

- **Notes on the Usage of the -autorun Option**

When using the `-autorun` option, it is important to note that everything following this option is treated as part of the third-party program.

Therefore, `-autorun` must be used as the last TclOK option, meaning other options like `-log` and `-verbose` should be placed before `-autorun`.

1. **Correct usage of -autorun:**

```
%  tclok test.tcl -log tclcheck.log -verbose -autorun run_tool
-f test.tcl
```

In this example, `-log` and `-verbose` options are placed before `-autorun`, and the third-party program `run_tool -f test.tcl` is executed correctly.

2. **Incorrect usage of -autorun:**

```
%  tclok test.tcl -log tclcheck.log -autorun run_tool -f test.tcl
-verbose
```

In this example, TclOK incorrectly considers `run_tool -f test.tcl -verbose` as the third-party program to execute, mistakenly treating `-verbose` as part of the program's options, rather than a TclOK option.

- **Using the -autorun_max_errors Option to Increase the Error Threshold for Automatic Program Execution**

By default, TclOK requires 0 errors for the automatic execution of a third-party program. If users want to tolerate a certain number of errors and still run the third-party program, they can specify the error threshold using the `-autorun_max_errors` option.

For example, if the user wants to allow the automatic execution of the third-party program when there are fewer than 3 errors in the Tcl script, they can execute the following command. As shown, since only 1 error occurred, which is below the set threshold of 3 errors, the `run_tool` program will still be automatically executed after the check completes.

```
%  tclok test.tcl -log tclcheck.log -verbose -autorun_max_errors 3
-autorun run_tool -f test.tcl
Error: can't read "value": no such variable
 Line: 1
 File: test.tcl
..................
****************** TCL CHECK SUMMARY ******************
  Total errors                : 1
  Total unknown commands      : 1 ->'set_parameter'
  Possible unknown commands   : 1 ->'hostname'
  Hierarchical summary        : test.tcl (errors:1, unknown_cmds:1
  ->'set_parameter')

FAILED, total error count '1' <= autorun limit '3', autorun is
launching 'run_tool ...'
..................
```

This option allows more flexibility by enabling the execution of the third-party program even if a limited number of errors are present in the script.

- **Using the -no_lic_enable_autorun Option to Run Third-Party Programs After License Checkout Failure**

Users often encounter situations where the license checkout fails, such as when they are unable to connect to the license server or forget to set the system variable `DASHTHRU_LICENSE_SERVER`, causing TclOK to fail to start and resulting in a final check status of `FAILED`.

By default, since the script was not checked, TclOK will return a `FAILED` result and will not automatically run the third-party program specified by the `-autorun` option. However, when the `-no_lic_enable_autorun` option is used, even if the license checkout fails, the third-party program will still run automatically. The usage is shown in the example below:

```
%  tclok test.tcl -log tclcheck.log -verbose -no_lic_enable_autorun
-autorun run_tool -f test.tcl
..................
Error: dial tcp 127.0.0.1:28000: connect: connection refused. (LIC-
  FAIL-INIT)
Error: license is not available, please confirm
  'DASHTHRU_LICENSE_SERVER' env has been set to the correct
  'port@host' which DashLM is running and serving. (LIC-CHECK-ENV)

FAILED, autorun is launching 'run_tool ...' because
'-no_lic_enable_autorun' is set
..................
```

Note: This option does not affect the normal behavior of the tool after a successful license checkout. After using this option, the third-party program will still be launched automatically based on whether the actual check result of the script is `OK` or `FAILED`.

4

4 TclOK Error Reporting Examples

TclOK has the following advantages over some open-source Tcl checkers:

1. Checks errors inside nested scripts
2. Checks complex errors in SDC/UPF
3. Adopts a "static + dynamic" check mechanism, effectively increasing the real error reporting rate and avoiding false positives

This chapter lists three error examples demonstrating these advantages of TclOK, covering the following content:

- Sub-Script Error Example
- SDC Error Example
- Eliminating False Positives Example

4-1 Sub-Script Error Example

TclOK not only checks errors in the top-level script but also hierarchically scans for errors in all sub-scripts. In the example below, the top-level script `top.tcl` executes three sub-scripts (Tcl, SDC, and UPF scripts), each containing a syntax error.

top.tcl

```
source design.tcl
read_sdc design.sdc
load_upf design.upf
```

design.tcl

```
if {$TESTMODE == true} {
        puts "Entering Test Mode"
        ……
}
```

design.sdc

```
create_clock -nane CLK -p 6 clk_port
…….
```

design.upf

```
set_level_shifter -domain PD \
-applies_to all \
-location self u1/in
……
```

In the TclOK verbose check report below, errors in all three sub-scripts are reported. Note that the commands executing sub-scripts might be built-in commands of third-party EDA tools, such as `read_sdc` in the example. If you need to check these sub-scripts, define these commands before TclOK starts checking. For example, simply define `read_sdc` as an alias for the `source` command, which can be done using the option `-x "alias read_sdc source"`.

```
%  tclok -x "alias read_sdc source" -verbose top.tcl
<info> line 1: sourcing script 'design.tcl' <---- current script
checking: if {$TESTMODE == true} ...
Error: can't read "TESTMODE": no such variable
 Line: 1
 File: design.tcl

<info> line 1: source script done, returning back to 'top.tcl' <----
  current script
<info> line 2: sourcing script 'design.sdc' <---- current script
checking: create_clock -nane CLK -p 6 clk_port
```

```
Error: wrong # args: unknown switch '-nane' for procedure
  'create_clock'
 Line: 1
 File: design.sdc

<info> line 2: source script done, returning back to 'top.tcl' <----
  current script
<info> line 3: sourcing script 'design.upf' <---- current script
checking: set_level_shifter -domain PD -applies_to all -location self
  u1/in
Error: wrong # args: unknown value 'all' for '-applies_to', should
  be one of following: 'inputs', 'outputs', 'both'
 Line: 1
 File: design.upf

<info> line 3: source script done, returning back to 'top.tcl' <----
  current script
****************** TCL CHECK SUMMARY ******************
  Total errors              : 3
  Total unknown commands    : 0
  Possible unknown commands : 0
  Hierarchical summary      : top.tcl (errors:0, unknown_cmds:0)
                            ├──design.tcl (errors:1, unknown_cmds:0)
                            ├──design.sdc (errors:1, unknown_cmds:0)
                            └──design.upf (errors:1, unknown_cmds:0)

FAILED
```

4-2 SDC Error Example

TclOK supports scanning deep errors in SDC scripts, such as missing options in `create_generated_clock` and filter expression errors in `get_*` commands shown in the example below.

```
design.sdc

create_clock -name CLK -p 6 clk_port
create_generated_clock -name CLK_DIV2 -source CLK -divide_by 2 -add d_reg/Q

set_input_delay 1 -clock [get_clocks CLK] [get_ports -filter "full_name = in*"]
set_output_delay 1 -clock [get_clocks CLK_DIV2] [get_ports -filter "is_output == 1"]
```

```
%  tclok -verbose design.sdc
checking: create_generated_clock -name CLK_DIV2 -source CLK -
  divide_by 2 -add d_reg/Q
Warning: potential wrong # args: '-master_clock' is required in
  some tools when '-add' is specified for 'create_generated_clock'
   Line: 3
   File: design.sdc

checking: get_ports -filter {full_name = in*}
Error: expect '==', '!=', '=~', '!~', '>', '>=', '<', '<=', '&&',
  'and', '||', 'or' or EOS but get '='
 Line: 6
 File: design.sdc

checking: get_ports -filter {is_output == 1}
Error: cannot convert '1' to boolean value
 Line: 7
 File: design.sdc
****************** TCL CHECK SUMMARY ******************
  Total errors              : 2
  Total warnings            : 1
  Total unknown commands    : 0
  Possible unknown commands : 0
  Hierarchical summary      : design.sdc (errors:2, unknown_cmds:0)

FAILED
```

4-3 Eliminating False Positives Example

The following shows a Tcl script with common errors, including a command argument error and a script file path error.

```
top.tcl

set a 123
if $a {
    puts $a $a $a;              # Incorrect usage of $a inline within puts
}
source bad_filepath.tcl;        # Incorrect file path
```

```
%  tclok top.tcl
Info: checked out license 'FusionShell-Lint'. (LIC-CO)

checking: puts 123 123 123
Error: wrong # args: should be "puts ?-nonewline? ?channelId?
  string"
 Line: 3
 File: top.tcl

checking: source bad_filepath.tcl
Error: couldn't read file "bad_filepath.tcl": no such file or
  directory
 Line: 5
 File: top.tcl

FAILED
```

By comparing the report results of TclOK and other Tcl checkers below, TclOK can effectively eliminate false positives, ensuring that errors occurring during the execution of third-party EDA tools can be precisely detected.

Tcl Script	Any EDA Tool	TclOK	Other Tcl Checkers
Line 2: Missing {} around expr	Pass	Pass	Issue a warning (Noise)
Line 3: Incorrect variable inline usage	Error/Stop	Error	Pass (Error Missed)
Line 5: Incorrect file path in source	Error/Not executed	Error	Pass (Error Missed)

A

Appendix A: TclOK Options List

% **tclok** **<tcl_file>**

Specifies the target Tcl script file to be checked.

[-autorun | -a] <ext_command> <options> ...

Automatically runs a third-party program when the check passes. All content following `-autorun` is treated as part of the external program. This option must be used as the last option.

[-autorun_max_errors <error_count>]

Specifies the maximum number of errors allowed before automatically running the third-party program.

[-verbose | -v]

Prints detailed log information during the check.

[-echo | -e]

Prints the script commands while checking.

[-log | -l] <log_file_path>

Specifies the log file path where the check results will be stored.

[-execute | -x] <startup_exec_script>

Specifies an initialization script to be executed. This option can be used multiple times to execute multiple scripts.

[-init <init_tcl_script_file>]

Specifies the initialization Tcl script file. Multiple script files can be specified by repeating this option.

[-cd <path>]

Specifies the directory to change into when starting TclOK.

[-lic_timeout <seconds>]

Specifies the license checkout timeout in seconds. Defaults to 2 seconds if not set.

[-no_lic_enable_autorun]

When a license checkout fails and causes the script check to fail, the third-party program specified by `-autorun` will still be executed using this option.

[-max_errors <error_count>]

Specifies the maximum number of errors allowed during Tcl checking.

[-max_report_unknown_cmds <unknown_cmd_count>]

Specifies the maximum number of unknown commands that can be reported during the check.

[-conditional_test_unknown_action <skip | as_true | as_false>]

Controls the behavior of `if`, `for`, or `while` conditions when the expression evaluates to unknown.
`skip`: Skips the entire statement.
`as_true`: Treats the expression as true and executes the statement.
`as_false`: Treats the expression as false and executes the statement.

[-loop_body_unknown_action <break | delay_break | go_on>]

Controls the behavior of loops (`for`, `while`, `foreach`) when encountering an unknown command inside the loop.
`delay_break`: Completes the current loop iteration before exiting.
`break`: Immediately exits the loop when an unknown command is encountered.
`go_on`: Continues executing all iterations of the loop even after encountering an unknown command.

[-strict | -s]

Enter strict check mode to report all unknown commands as errors. It is recommended to define all third-party tool commands in the Tcl script as known commands before using this mode.

[-**strict_upf** | -**u**]

Ensures that no non-UPF commands (such as `get_cells` or `get_pins`) appear in `.upf` files or files loaded by `load_upf`.

[-**stdin <text>**]

When `gets stdin` or `read stdin` is encountered in the script, the check pauses to wait for user input. Use this option to specify the text to be used as input, preventing the check from pausing. This option can be used multiple times to correspond to multiple `stdin` inputs in the script.

[-**enable_ext_exec "command_pattern1 command_pattern2 ... "**]

Executes external commands during the check process using `exec`.

[-**print_var_at_line "(filepath_pattern:)line var_name1 var_name2 ..."**]

Prints the values of specified variables at specified line numbers. This option can be repeated to print variables at multiple locations in the script.

[-**no_utf8**]

use non UTF-8 characters when printing the hierarchical file tree report.

[-**version**]

Prints the version number of TclOK.

[-**help** | -**h** | --**help**]

Prints the help information for starting TclOK, including available options.