An Implementation of Comprehensive Static Analysis of Tcl Syntax and Scoping

Justin Egli
Mentor Graphics Corporation
8005 SW Boeckman Rd.
Wilsonville, OR, 97070
justin_egli@mentor.com

Sridhar Srinivasan
Mentor Graphics Corporation
8005 SW Boeckman Rd.
Wilsonville, OR, 97070
sridhar_srinivasan@mentor.com

Robin Albrecht
Mentor Graphics Corporation
8005 SW Boeckman Rd.
Wilsonville, OR, 97070
robin_albrecht@mentor.com

ABSTRACT

In our tool framework, Calibre PERC [1], we use Tcl as the command language. Users write rule decks in Tcl to perform complex electrical and design rule checks. Large rule decks can take several hours to days to run; due to the interpreted nature of Tcl, it is difficult for the deck writer to debug simple but fatal errors in the Tcl rule deck, such as invalid variable usage, invalid global variable usage across different interpreter scopes, illegal use of commands and other scoping errors. To mitigate these mistakes we wanted to extend a Tcl static analyzer to identify these errors before they are encountered in the field. In our paper we will describe how we identify these problems by static analysis.

1. INTRODUCTION

Modern silicon manufacturing and design technology involves multiple teams from different companies coming together. The EDA companies develop tools that design houses use to create the design and use foundry process and design development kits to verify their designs. Our tool framework is used in physical and circuit verification flows by design companies with process rule decks created by manufacturing foundries like TSMC, IBM, Global Foundries, etc., In Figure: 1, Calibre is the physical verification tool provided by EDA company Mentor Graphics, Design Database is owned by design house that is building the design, Foundry PDK, runsets or rules are provided by the foundry where the design is going to be manufactured. The rule decks that work with our physical verification tools are put together using our proprietary control language which includes Tcl (Figure: 2). Usually these rule decks are encrypted to protect foundry IPs. Since Tcl is an interpretive language [2] [3] rule deck errors can occur in the field, where it is cumbersome and expensive in terms of both man power and time to fix these design kits. Typical errors that are encountered in the field:

- Syntax errors like uninitialized variables and use of variables without declaring: Since Tcl is an interpretive language, it is possible that certain variables get initialized in loop or branching code that may not be executed in the field.
- Invalid global variable use across interpreters: To enable encryption and multi-threading, our tool framework will instantiate new Tcl interpreters, global variable use across these interpreters are invalid.
- Syntax errors in custom Tcl commands: These are caused by invalid command options or invalid use of Tcl commands that are part of our APIs.

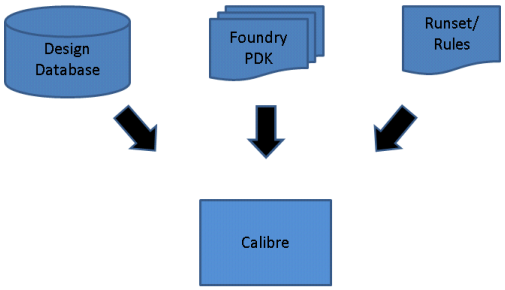

Figure 1: Typical Tool Flow Setup.

2. EXISTING METHODOLOGIES

In the tool flow explained in section 1, during the design stage the design database is verified using the verification tool and the runset provided by the foundry at the design house. In cutting edge designs, the runset, verification tool features and the design are all in development concurrently. The foundry that provides the runset usually encrypts the content to protect their process IP. If there is an error in the runset, then currently they have to debug that in the field by providing multiple versions of the runset with some print statements. This can be very tedious and time consuming. To prevent potential field issues, the runset developer usually creates a lot of testcases to ensure that his rule deck is performing the checks appropriately. If the testcase does not cover all possible scenarios then there can be syntax error hiding in the runset, ready to explode at the design house.

2.1 A Typical Rule Deck Error Scenario

Refer to Figure: 3, it shows an command syntax error hiding in the section analyzing the capacitances. If the runset quality assurance test suite did not exercise the runset with a testcase containing a capacitor, then this will never be caught during regression testing. When an error like this shows up at the design house, since the runset is encrypted

Figure 2: Runset/Rules.

it needs an expert from the foundry team to debug it. To make this worse the design house and foundry are from different continents.

In this paper we present a Tcl static analyzer that can check errors early on, where the runset developer could use it as a linter to catch these bugs before the runset get shipped to their customer, avoiding expensive debugging time.

3. OUR IMPLEMENTATION

We first began with the hopes of finding a static analysis tool that would already check for some of error cases we were most worried about. The first case we were worried about was the use of uninitialized variables due to potentially changed execution paths. Unfortunately, none of the tools we evaluated successfully checked this simple case. It soon became apparent that we would have to add most of the functionality we wanted ourselves. With this information in mind, we chose an open source Tcl static analyzer called Nagelfar as our starting point. Nagelfar offered the most complete error checking and was a good initial framework for our use [4]. The following is a list of functionality that we wanted in our analyzer.

1. Support for the Tcl "source" command.
2. Tracking use of global variables and identified potential use of un-initialized variables involving control structures such as if-then-else constructs.
3. Command and function dependency graph and checking call groups.

To detect these error messages we first needed to make more detailed information available on the scope of variables and add support for evaluating code beyond a single file. Most of our customers opted to organize their rule

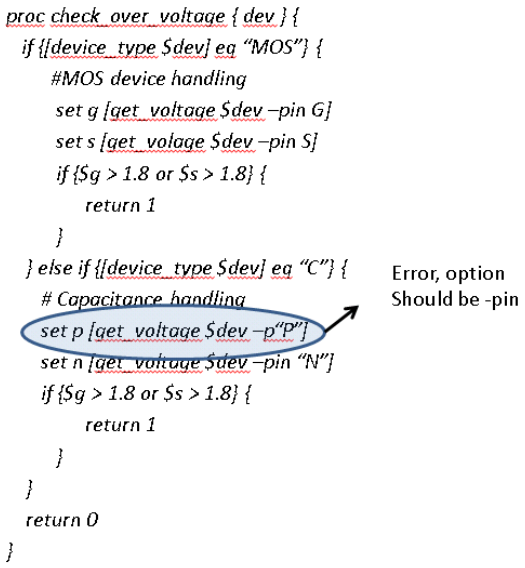

Figure 3: Command Syntax Error Hiding.

decks in different files and include them with the source command. Further analysis targeting variable usage in control constructs was developed to track the instantiation and usage of variables in conditional constructs like if statements. Variable references needed to detect the proper namespace that was being accessed for later evaluation against illegal shared access.

3.1 Tracing Through Included Files (Tcl Source Command)

The first feature that we added was the ability to follow the Tcl "source" command and analyze the specified file. This is a fairly simple step in the analysis process; upon encountering the command in a file, the analyzer only needs to open the file and evaluate it in place. The behavior of the source command is fairly easy to replicate. It searches for the file in the context of the position of where the script began. An example is provided below:

```
set filePath [lindex $args 0]
set firstDir [lindex [file split $fileName] 0]

if {$firstDir != "/" && $firstDir != "~"} {
    # Join source path with path of the current
    #file, parseFile returns the initial file
    #specified for parsing

    set filePath [file join \
                        [file dirname [parseFile]] \
                        $filePath]
}
if {[file exists $filePath] && \
        [file readable $filePath]} {
    # Begin in place parsing of this file

    parseSubFile $filePath $currentState
} else {

    analysisMsg error "Couldn't parse file $filePath"
```

```
}
```

The important part of handling most source commands is correctly recognizing the path that should be used to find the file. While the above code will handle most cases, there are ways of changing the current directories in Tcl, where this approach would not work. One way to cover these cases is to add an argument the parser recognizes that allows one to specify paths. Then expand this code so it searches from the path folders to find the specified file. This simple and robust way of evaluating the effects of source commands provides us with the ability to handle projects that organize their code over multiple files.

3.2 Global and Namespace Variable Scoping

To implement some of the checks regarding variable usage we needed to expand the amount of information the analyzer was tracking. In particular it needed information on the scope in which these variables were used. It was not enough to split up variables by their function scoping as was being done previously. We extended variable handling to fully resolve the namespace and include namespace information. We also expanded the function parsing to keep track of when arguments were used to create variables in a namespace. This way, variables defined dynamically by namespace functions would be tracked.

To resolve the variable name, we needed to be aware of the local scope it was in. If it was in a function, then the variable does not reference any scopes outside of the local scope unless it specifically has a namespace specifier in it, or if it is created via the use of the namespace command. Outside of functions, any variable reference is in the context of some namespace. We can resolve these by keeping track of the current namespace we are executing in. Any variables without a namespace specifier in it will be "set" in the current namespace, but if they are used, we first search the current namespace if they are defined, then look at the global namespace. This is because Tcl will fall back to referencing the variable in the global namespace if it isn't defined in the current one. Furthermore, we need to fall back to this behavior when dealing with relative namespace references. A relative namespace reference is when there is no namespace specifier "::" at the front of the reference, these references will treated just as the regular references are. Sets will always set in reference to the current namespace, reads will check the current namespace, then check the global namespace if nothing exists in reference to the current one.

```
# Returns a list of one or two entries.
# If there is only one entry, that is the
#only possible reference. If there is two,
#then there is two possible paths with the
#first getting the highest priority.
proc getFullVariable {var {isSet 0}} {

    if {[currentProc] != "" && \
            ![isNamespacePath $var]} {
        # Variable local to a function

        return [list $var]
    } else {

        # Not in function or has a namespace
        #specifier

        if {[string first "::" $var] == 0} {
            # Global specifier, only one
            #possible reference

            return [list $var]
        }
        # Some kind of relative path; Get
        #current namespace

        set ns [currentNamespace]
        if {$ns == "" || $isSet} {
            # In global Namespace, only
            #return path relative to global
            # OR Set operation, always refer
            #to path relative to current ns

            return [list "${ns}::$var"]
        }

        # A read operation return two possible
        #paths, relative to local then global
        return [list "${ns}::$var" "::$var"]
    }
}
```

The analyzer still needed more information to completely understand when uninitialized variables were potentially being used; in particular, it needed to know when these variables were being used in control structures. A stack was added that keeps track of entered control structures; when a control branch is entered, such as an if, elseif, or else statement, the branch is assigned a unique id and a combined group id that describes the entire structure. When variables are set we can look at the current control we are in and add that information to the variable. When a variable is used, we can check to see if the control information of the variable being accessed is still in the control stack. If the unique id is in the stack, the variable is being accessed in the same control, or one of the controls that is a child of it, and we know it has been set. If the id isn't in the stack, we know that it has been used in a situation where it might not have been set and we can generate a warning or error.

If the variable is set in all possible branches, such as when an 'if' construct ends in a 'else' statement and each branch has a set for the variable, then we know that we should treat the variable as if it is always set. Some extra information needs to be tracked in the controls to accomplish this, there needs to be an index that increments for each control in the same control group and a flag indicating if the control is an else statement. Each parsed set operation can add to a list of branches associated with the control group for the variable being set. When the final else statement is encountered, the index can be checked against the branch list to see if it is set in each branch. If it's set in all the branches, the variable will then be set in the parent structure. Unless there are no more parent control structure, in which case it will be considered set for the rest of the global or function scope it is in.

```
# setVariableInControl is only run when
#the variable to be set is not set and
#the parser is in a control.
proc setVariableInControl {var} {
```

```
set control [currentControl]
set i 0

while {1} {
    set cntrlId     [lindex $cntrl 0]
    set cntrlGrpId  [lindex $cntrl 1]
    set index       [lindex $cntrl 2]
    set isElse      [lindex $cntrl 3]

    lappend parsedVars(cntrl,$var) $cntrlId
    lappend parsedVars(cntrl,$cntrlGrpId,$var)\
        $cntrlId

    # If we are an else statement, we are
    #at the end of the structure
    set branchCount \
        [llength $parsedVars(cntrl,$cntrlGrpId,$var)]
    if {$isElse && branchCount == $index)} {
        # Check if we have set the variable
        #as many times as there are branches
        #in this controlGroup. If we have it
        #has been set in each branch.

        array unset parsedVars \
            "cntrl,$cntrlGrpId,$var"

        if {([llength $::cntrlStack] - $i) == 1 } {
            # Top entry in control stack;
            #setting in parent means we are
            #set in the global/local scope.
            # Stop tracking as a control variable
            array unset parsedVars "cntrl,$var"
            return
        } else {

            # Set in parent control structure
            #and loop around
            incr i
            set cntrl \
                [lindex $::cntrlStack end-$i]
            continue
        }
    }
    # Finished setting control info.
    return
}
}
```

Checking usage of variables against this control info can be accomplished by the following:

```
if {[info exists parsedVars(cntrl,$var)]} {
    foreach setCntrlId $parsedVars(cntrl,$var) {
        foreach cntrl $::cntrlStack {
            if {$setCntrlId == [lindex $cntrl 0]} {
                return exists
            }
        }
    }
    return maybe
}
```

If the variable has control information attached to it, check if we are in one of the controls that set it. If we are, it is accessible. If it isn't, it may or may not be accessible depending on which branches may have been taken.

3.3 Call Graphs and Threading

To identify and prevent shared global usage across interpreters, the analyzer needed to know which function calls refer to global variables and how these variables are accessed. We began with a simple call graph. As each function is parsed, an array entry is created for the function with a list of each function it calls. This creates a graph where the 'top' node is the global context. We can now use this graph to organize functions into call groups, which is a grouping of functions by threads. Given a list of functions that act as entry points for separate threads, we can build this graph. Any nodes that follow these entry points and have no parents that are from a different group can be assigned to a specific call group. By making use of the improved namespace handling we can create an array that maps each function to all the global variables they use. Any global variable accesses by the call group functions are specific to that thread and these variables can be marked as belonging to that thread. When we come across a variable that has already been used by a different thread, we can issue a warning against shared usage across threads. Also, any functions which are shared between multiple call groups will generate warnings for any global variable usage within.

```
# Adds a call to the callgraph, called everytime
#a command is parsed.
proc addCall {cmd} {

    # Get the calling function
    set caller [currentProc]
    if {$caller == ""} {
        # Instead set parent as calling namespace
        set caller [currentNamespace]
    }
    if {![info exists $::CallGraph($caller) ||
            [lsearch -exact $::CallGraph($caller)\
                $cmd == -1]} {
        lappend ::CallGraph($caller) $cmd
        lappend ::CallGraph($cmd,parents) $caller
    }
}

proc parseGraph {groupProcs} {
    # CallInfo contains information on which
    #group(s) a function belongs to
    # CallInfo($group) List of all functions
    #belonging to $group
    # CallInfo(shared) List of functions shared
    #between groups
    # CallInfo(known,$proc) Group $proc belongs
    #to (if not shared)

    array set ::CallInfo [list shared {}]
    foreach group $groupProcs {
        lappend ::CallInfo($group) $group
        set ::CallInfo(known,$group) $group

        foreach proc $::CallGraph($group) {
            parseGroup $group $proc
        }
    }
```

```
    checkGlobals
}

# Parse callgraph and organize functions by group
proc parseGroup {group proc {sharedParent 0}} {
    # If we encounter a shared function we can
    #stop traversing this node, it has already
    #had all it's children marked.
    if {[lsearch -exact $::CallInfo(shared) $proc]\
             != -1} {
        return
    }

    # Anything with a shared parent is also a
    #shared function
    if {$sharedParent} {
        unset -nocomplain ::CallInfo(unknown,$proc)
        lappend $::CallInfo(shared) $proc
        # Mark all child functions as shared too
        foreach leaf $::CallGraph($proc) {
            parseGroup $group $leaf $sharedParent
        }
    }

    # Check to see if parents belong to same group
    foreach parent $::CallGraph($proc,parents) {

        # If Parent matches group, continue
        #checking parents
        if {[info exists \
                ::CallInfo(known,$parent)]} {
            if {$::Callinfo(known,$parent) \
                    == $group} {
                continue
            }
            # Shared;mark all descendants shared
            parseGroup $group $proc 1
            return
        }
        # Parent is unknown, stop checking and
        #try again when parent is parsed
        return
    }

    # Known and in group
    lappend ::CallInfo($group) $proc
    set ::CallInfo(known,$proc) $group

    # Parse child functions
    foreach leaf $::CallGraph($proc) {
        parseGroup $group $proc
    }
}

# Check global usage against graph info and see
#if any are shared
proc checkGlobals {} {

  # Start by looping through each global variable
  foreach var [arrayName ::GlobalsUse] {
    set usedIn ""

    # Check each function that uses it
    foreach proc $::GlobalsUse($var) {

      # Usage in shared functions is
      #automatically marked as an error
      if {[lsearch -exact $::CallInfo(shared)\
          $proc] != -1} {
        analysisMsg "Global $var used in\
                    shared function $proc"
        continue
      }

      # If used in a group, check group
      #versus previous usage group
      if {[info exists \
          ::CallInfo(known,$proc)]} {

        set group $::CallInfo(known,$proc)
        if {$usedIn == ""} {
          set usedIn $group

        } elseif {$usedIn != $group} {
          analysisMsg "Global $var used in $group\
            ($proc) after use in $usedIn"
        }
      }
    }
  }
}
```

4. ISSUES

These examples shown above were done as a proof of concept and as a learning tool. There are some cases where they may not work properly or cause errors. Some of the array keys could be corrupted depending on the naming of variable or functions in the user's scripts. The array `parsedVars(cntrl,$var,$id)` could have values overwritten if the script contains variables named "a" and "a,1". We make the assumption that reasonable variable naming conventions are followed by the script writers.

Other issues include dealing with possible obfuscation or other redirections in Tcl. Our callgraph and function group handling code does not handle renamed functions for example.

5. CONCLUSION

We have presented an implementation of several static analysis algorithms for use with Tcl. We covered the current state of static analysis in the Tcl community and detailed ways in which to expand it. We tried to highlight the benefits of using static analysis in scenarios where testing with an interpreted language can miss simple mistakes. Currently, most testing and debugging approaches in the Tcl community are manual, adding 'puts' statements to see information; we believe good static analysis tools can make this process quicker, more thorough, and will improve productivity.

References

[1] Mentor Graphics Corp. *Calibre PERC User's Manual.* Mentor Graphics Corp, Wilsonville, OR, 97070, 2014.

[2] Brent B. Welch; Ken Jones. *Practical Programing in Tcl and Tk*. Prentice Hall, Upper Saddle River, NJ, 07458, 2003.

[3] Volunteers. Tcl developer xchange, http://www.tcl.tk/man/tcl8.4/, 2005.

[4] Peter Spjuth. Nagelfar, http://sourceforge.net/projects/nagelfar/, 2014.