# Comit's CVXT Tool

**Venkat Iyer**
**Comit Systems, Inc.**

# 1. Abstract

CVXT is a Verification Engine from Comit that dates back to the early 90s. It is primarily an control and event based interface between a hardware language software simulator and tcl. This paper is about the Tcl and software side of the tool, the challenges it faced and how coroutines helped.

# 2. Introduction

## 2.1. About Comit

Comit is a 17+ year old silicon valley company. Our main business has been designing chips, boards, systems and software for our customers. We call it contract engineering as we go all the way from specifications to supporting volume production of chips. All along we have developed tools that improve productivity and improve hardware design effectiveness.

## 2.2. Comit and Tcl

From web servers to invoicing to physical layout tools - everything happens in Tcl at Comit. We have a whole set of design tools code-named Fiesta® for Process Standardization and Accelleration. Most of these tools target areas not traditionally addressed by standard industry EDA tools. Many of the tools are completely Tcl, but all use Tcl to some degree.

## 2.3. CVXT

One of these tools is called CVXT, the verification engine. We wrote this tool in the early 1990s to be able to write tests in Tcl to verify hardware designs.

# 3. HDL Simulation

A Hardware Description Language (HDL), like Verilog or VHDL, is used to describe digital hardware circuits. Boolean gates like AND and OR are most familiar to software engineers. But designing hardware in gates would be like writing software in machine code. HDLs provide a far higher level of abstraction, akin to what C would be to machine code. This HDL can be then "synthesized" into digital logic which is put into silicon. This HDL can also be verified by simulation.

HDL simulators are essentially event driven scheduling kernels, which have lots of virtually parallel processes. Each process can suspend for some simulation time or an

event. HDLs are still targeted for designing hardware, and they don't lend themselves to easy test writing. Over the years, HDLs themselves have added features to make writing tests and building test environments easier. SystemVerilog is an example.

Like CVXT there have been attempts to build interfaces from Verilog to other languages, Vera (from Synopsys) and Specman/E (from Cadence) are two examples. There have also been attempts to do Hardware Design using more common languages - JHDL in Java, and SystemC in C++ are two examples.
But Verilog (mostly in the US) and VHDL have remained the mainstay for digital design. At Comit, more than 90% of the designs we do are in Verilog.

# 4. Why Tcl in CVXT

The early reasons for doing CVXT were to make our verilog simulations more realistic. One example is to display Images before the hardware processed it, and as it was processing it. Another example is to snoop packets off the network and send them for processing into the hardware design.

The main reasons for using Tcl:

- Tcl is built to be embedded, so building it into multiple simulators would be easy
- Tcl supports event driven programming. Though I don't use the tcl event loop, most of the support infrastructure is used here.
- Tcl is interpreted and dynamic
- Tcl provides access to OS and hardware features in a platform independent way
- Tcltest provides a test framework

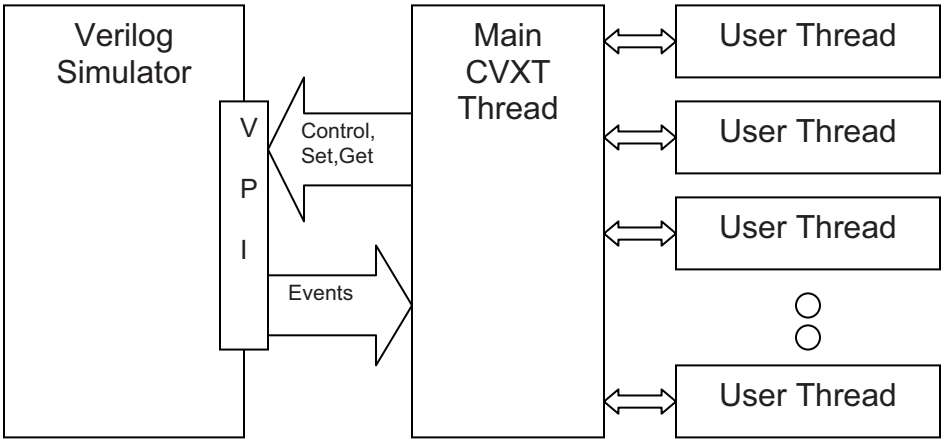
And pleasantly, Tcl is a known language in the hardware design world. As most users of these tools are hardware designers. Most Electronic Design Automation (EDA) tools have moved to using Tcl as the scripting as well as the interactive command language. Mentor Graphics was an early adopter in vsim, but now it's hard to find a non-Tcl tool.

# 5. Brief CVXT Description

CVXT provides the hardware verification engineer with a multithreaded, event driven, test environment, to effectively verify the hardware design before committing it to silicon.

This paper talks about CVXT from the software point of view. The user, a hardware verification engineer, writes his tests in Tcl, over the layers of libraries. The interface to the simulator, the kind of simulator is transparent. Sometimes the same test is run over real hardware rather than a simulator.

At the lowest level, this is how CVXT allows you to run tcl inside a hardware simulator. Verilog has a standardized C API called VPI (or sometimes PLI2), which all the simulators support.




As the figure shows, there are two modes of communication between Tcl and Verilog. There's a synchronous interface shown as the arrow going right to left.
This is mainly how the CVXT thread is started up, and how tests in Tcl can set signal values in verilog or get hardware signal values from the simulator, and how Tcl controls simulation progress.

The Asynchronous interface is shown as an arrow going left to right. These are callbacks from the verilog simulator. There are two main kinds of callbacks. One is value change callbacks (VCBs), which are like variable write traces. The other is a time callback, where you can be called back after a particular amount of simulation time.

# 6. Some CVXT operation details

I use the word "thread" loosely. The Challenges section below will elaborate more on that. There is one main controller thread that manages and synchronizes user threads. Each user thread runs a tcl script, with access to a set of facilities provided by CVXT to interact with the hardware program.

The threads themselves are cooperating threads, which is very much like HDL processes. Each thread runs till it's done for now and wants to sleep till some specified event. Like HDL processes, the threads are perceived to run in parallel, but are run one-by-one.

## 6.1. A Simple CVXT Example

```
set r [get tb.ethernet0.error]
if {$r == 0} {
   put tb.ethernet0.txen -value 1
   set w [wait -signal tb.ethernet0.eof -time 1000]
   set tb.ethernet0.txen -value 0
   if {$w eq "time"} {
      error "Packet was not transmitted in 1000 ns"
   }
}
```

The get retrieves the value of a hardware signal. The put changes the value of a hardware signal. The wait statement says that we are done for now and are waiting for a signal to change or a timeout to occur.

The interesting part is that when the wait is encountered, this thread suspends. Other threads continue. When the simulation proceeds to a point where the eof signal changes, this thread continues.

# 7. The Challenges (and solutions)

The main source of the challenges were the variety of simulators and OSs to be supported. The tool is deployed over a variety of simulators, e.g.

- Aldec's Riviera Pro (with Tcl)
- Cadence's NC Verilog (with Tcl)
- Mentors Modelsim (with Tcl)
- GPL Cver (no Tcl)
- Icarus Verilog (no Tcl)

For each simulator, various versions with each's own idiosyncracies are supported.

The operating systems supported are

- Linux 32-bit
- Linux 64-bit
- Solaris 32-bit
- Solaris 64-bit
- Windows 32-bit

## 7.1. Threading

Each user thread expects to retain context. The code after the wait should run just as if it had just paused there.

This translated nicely into Tcl's threaded model. Early tool versions used Tcl built with threads enabled. Each user thread was mapped into it's own Tcl Thread in it's own interp. Context switches were handled as thread switches. This worked well on Windows, but on Unix, we had a lot of hard-to-explain crashes, depending on whether other simulator extensions were loaded. We traced this down to libraries being different on pThread based implementations, and conflicting libraries getting loaded - some threaded, some non-threaded.

We then moved to using ucontexts. We still built Tcl threaded, because it had to handle context switches, but we didn't use the Tcl thread extension. On windows, we just built a ucontext library which was just a wrapper around real threads anyway.

Inter-thread communication was already handled as a single argument during invocation and resumption. A thread suspension/exit causes a single (complex) return value, which results in a long

## 7.2. Multiple Tcl Versions in one process space

I hit this problem many times in many variations. The EDA vendors started favoring Tcl, but they remained woefully behind. The latest and greatest from one EDA vendor is on 8.4.13. I support many versions. I only dropped support for 8.0.5 in the simulator last year.

On the other hand, CVXT has stayed current (often going off cvs head, as it does now). I like my users to have access to the latest and greatest Tcl features. Object orientedness and dicts being the ones I needed Tcl 8.5 mostly for.

### 7.2.1. Build Process

I build Tcl from sources with threads enabled. I leave the source directory lying around with the objects.

```
% ./configure --enable-threads -enable-shared
% make
```

I do not run the make install step. At the end I have all the objects build for the full Tcl library, including a few that I don't really need - like tclAppInit.o and cat.o.
The cvxt files are compiled with -DSTATIC_BUILD, which only affects Windows. The link is also special. Logically:

```
* gcc -m64 -Wl,-Bsymbolic -o cvxt.so <cvxt objects> \
    <tcl core objects> -shared <platform specific -l flags>
```

The -Bsymbolic forces gcc to bind global references locally, thereby preventing cvxt from calling functions in the Tcl version built into the simulator.

### 7.2.2. Run Time

cvxt.so is loaded into the simulator (e.g. with the -loadpli flag on Riviera Pro). At startup it adds commands that can be used in vhdl, and optionally starts up some tcl threads. Threads are created on the fly from Tcl and or HDL, and die on error or finishing execution.

Extensions to cvxt in C are all stubs enabled, and are blisfully unaware of the multiple Tcls.

# 8. Enter Coroutines

Coroutines were a wonderful fit. I started using them as soon as they got into the sourceforge codebase. My previous thread implementations were in C. Now I could handle the threads at the script level. Here are the major advantages for me to switch to coroutines:

No more threads. I could build Tcl with --disable-threads and still be able to handle context switches. This

- simplified my build process
- sped up creation of threads and context switching

The possibility of having multiple contexts per thread. I call this branches. This simplifies datasharing and is very similar to fork and join in Verilog.

I've already deployed versions with coroutines transparently to users.

# 9. CVXT Coroutine Implementation

Tcl is started off by a call to the thread creation routine from either the HDL or from the simulator as part of it's initialization.

## 9.1. Thread Creation

On the first call, cvxt initializes the tcl subsystem, creates the main interpreter, and forks off the user thread based on the parameters passed in.
The user tcl runs all the way to a wait or till the end. The wait causes the thread to suspend (yield). When the main thread wants to wake the thread up, it sends the reasons for the wakeup to the thread, which becomes the return value of the yield. The following "code" shows the creation procedure in the main interpreter.

```
proc create_thread args {
  set u0 [interp create u0]
  foreach cmd [list get put] {
   $u0 alias $cmd $cmd
  }

  $u0 eval {
    proc wait args {
      return [yield [concat u0 $args]]
    }

    proc start {} {
      catch {
       user code here sourced from $args.
      }
    }
  }
  process [$u0 eval coroutine __run__ start]
}
```

## 9.2. Thread Switching

The process procedure in the main thread expects to get a list of events that the thread wants to wait on. The main thread adds callbacks into the simulator, processes. If any other threads have been activated, it runs them. When no threads remain, it returns control to the simulator.

```
proc process l {
  set child [lindex $l 0]
  foreach arg [lrange $l 1 end] {
    add call backs into simulator, and remember in data structure
  }
}


proc callback args {
  # called by simulator
  figure out which threads need to be awakened
  foreach interp $wakeup_threads {
     process [$thread ::__run__ $args] ;# resume the coroutine.
  }
}
```

## 9.3. A more complex example

This example is of a mailbox between multiple user threads. This is again common in EDA verification environments. Each mailbox can have any number of producers (writers), but only one consumer.

```
class create mailbox {
   constructor {} {
      variable name
      set name [namespace tail [self object]]
      if {[catch {set mbdata $shvar(mailbox_$name)}]} {
        set mbdata [list]
      }
      set shvar(mailbox_$name) $mbdata
   }

   method put args {
      variable name
      lappend shvar(mailbox_$name) $args
   }

   method get {} {
      variable name
      while {![llength [set mbdata $shvar(mailbox_$name)]]} {
        wait -shvar mailbox_$name
      }
      set shvar(mailbox_$name) [lrange $mbdata 1 end]
      return [lrange $mbdata 0]
   }
}
```

This is akin to having one server with multiple clients. The server thread serves register access requests. It waits for a request on the writer queue, does the transaction and puts the response back on the response mailbox.

```
mailbox create  reg_writer
while 1 {
  lassign [reg_writer get] mbox op addr data
  do the register transaction in simulator or hardware
  mailbox create $mbox
  $mbox put $read_data
}
```

The client thread looks like this:

```
mailbox create  reg_writer
mailbox create  reg_response
reg_writer put reg_response read 0x30943544 0x33
set reg_value [reg_response get]
```

# 10. Future Directions

Implement branches. This allows for multiple contexts in one thread, but with shared globals.

Package as a single file shared lib, one readme and and the tclsh replacement, to allow the framework to work outside of a simulator.

# 11. Acknowledgements

John Osterhout for the Tcl Language
Activestate, the TCT, maintainers and supporters of the Tcl Language and the Wiki
Miguel Sofer for NRE.
Many Tclers in the Chatroom, who've helped me with various details
Organizers of Tcl2009.

# 12. About the Author

Venkat holds a degree of Master of Technology in Computer Science from Pune University, India. His work interests are Logic Design, Tcl, Languages and Automation. He has contributed to many open source projects and enjoys basketball, snowboarding, playing the guitar, and hiking. He leads the engineering at Comit Systems, Inc. More information about him is at http://wiki.tcl.tk/vi. He can be reached by email at venkat@comit.com