Creating Virtual Peripheral Devices in a Digital Circuit Simulator Using Tcl/Tk

Jeffery P. Hansen
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

TkGate is a digital circuit editor and simulator built as a hybrid C and Tcl/Tk application. It is used in circuit design courses at dozens of universities throughout the world and has interface support for nine European and Asian languages. In this paper we primarily focus on a feature of TkGate called Virtual Peripheral Devices (VPD). VPDs are simulated representations of digitally controlled physical devices or systems. A VPD has a graphical interface, implemented in Tcl/Tk, representing the device and a Verilog stub module representing the digital interface to the device. Users interact with the simulated device through the GUI while a Verilog description of the control circuit interacts through the VPD stub module.

1 Introduction

TkGate [1] is a digital circuit editor and simulator designed to be used in teaching digital circuit design at the university level. It provides a wide range of built-in features and is suitable for design up to and including medium sized microprocessors. TkGate was originally designed and built as a non-Tcl/Tk application and later ported to use Tcl/Tk. Tcl/Tk proved to be powerful and versatile enough to make this a surprisingly painless port. TkGate has since continued to use Tcl/Tk as an integral part of the application.

One interesting feature of TkGate, and the focus of this paper, is the ability for users to define Virtual Peripheral Devices (VPDs). VPDs are Tcl/Tk scripts that simulate a peripheral device or other physical system and include both user interaction and interaction with a controlling circuit description. A typical use for VPDs would be to create engaging and realistic laboratory assignments for University level engineering courses. Two example VPDs are included with the standard TkGate distribution: a TTY device, and a drink vending machine. These will be further described in Section 3.

TkGate handling of VPDs is designed to be user extensible. Users create a VPD by writing a Tcl script to define the physical behavior of the device, and a Verilog library file containing a stub module that interacts with the Tcl script. Possible VPDs could include not only traditional devices like terminals and printers, but also any digitally controlled system such as a fly-by-wire controller for an aircraft, a train controller, etc.

This paper is organized as follows. Section 2 will give a brief introduction on the history and architecture of TkGate. Section 3 will describe the two VPDs that are included with the TkGate distribution. Section 4 will describe the Tcl-side and Verilog-side API for writing new VPDs. Section 5 will discuss the implementation of VPD handling in the simulator. Finally, Section 6 will summarize the paper.

2 TkGate History and Architecture

Work on the predecessor to the TkGate digital circuit editor and simulator, Gate, was started in 1986. It was designed for a window manager called "wm" which was developed as part of the Andrew project at Carnegie Mellon University. As X11 gained popularity, it was then ported to run under X11 as an Xlib application in about 1990. Finally, in the late 1990s, TkGate was ported to run as a hybrid C and Tcl/Tk

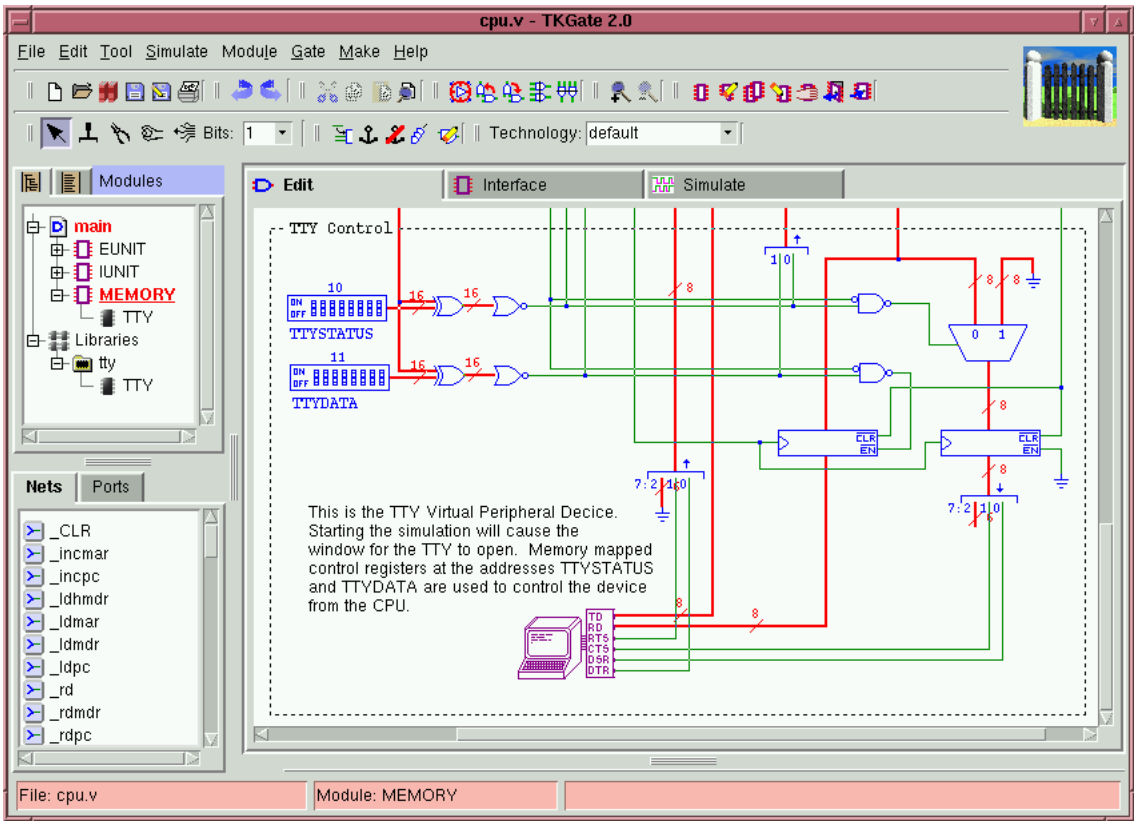

Figure 1. TkGate Interface

application.

The port to Tcl/Tk was performed by treating the XLib-based main editor window in Gate as a Tcl/Tk user-defined widget through the C interface, and converting all of the other interface elements such as menus and dialog boxes to use Tcl/Tk. Using this approach it was surprisingly easy to port Gate from a pure XLib application to a Tcl/Tk application and an initial running version was possible with only a few days of effort. The first public release of the simulator was TkGate 0.9 in May 1999. The current publicly released version is 1.8.6, and version 2.0, which contains support for VPDs, has an expected public release by the end of 2005 (Developer snapshots are currently available on the TkGate web site[1]).

2.1 Verilog Simulation

TkGate uses a simulator based on Verilog[2]. Verilog is a highly popular Hardware Description Language (HDL). Many commercial tools used for engineering circuit design including design entry, simulation, synthesis and verification are based on Verilog. A Verilog design is organized as a collection of "modules". A module can have one or more inputs, outputs, and bidirectional ports. Verilog supports design of structural (or netlist) modules, behavioral/dataflow modules, or a combination of both. Structural/netlist modules are defined as a set of module instances and wires that connect their ports. Behavioral modules are defined using programming constructs similar to those in C.

Unlike C, Verilog has an explicit concept of simulation time. Simulation time is the time at which events such as changes in a net or register value occur in the design being simulated. It is usually measured as an integer number of discrete time units. An event-based simulator such as that used by TkGate, keeps track of the time at which events occur in simulation time. Most operations in Verilog have a way of specifying a time delay in these time units.

Verilog is designed to support parallelism. A typical behavioral Verilog description is comprised of many

parallel threads. Most of the threads are small loops which emulate some aspect of the hardware behavior. Strictly speaking, on a uniprocessor machine, while the threads execute in parallel with respect to the virtual simulation time, the threads do not literally execute in parallel, but execute sequentially in such a way as to keep the simulation time among the threads in lock-step.

2.2 TkGate Structure

TkGate is comprised of two main executables, a user interface (TkGate itself) and an event-based Verilog simulator called Verga. The interface (shown in Figure 1) is comprised of about 55,000 lines of C and 26,000 lines of Tcl/Tk. All direct user interaction is through the interface which includes circuit editing and control of the simulation. The simulator runs as a separate executable and includes extensions to support VPDs.

TkGate has two main modes: an "edit" mode and a "simulate" mode. While in "edit" mode, the user can modify the circuit data through a graphical interface. TkGate supports both graphical modules displayed as components connected by wires, and Verilog text modules edited through a built-in text editor. When saving circuit data, or transmitting it to the simulator, graphical modules are saved in Verilog netlist format with annotated comments to indicate the screen position of devices and wires.

While in "simulate" mode, the user can set breakpoints, invoke script files, set/remove probes and query signal values. Communication between the TkGate interface and the Verga simulator is through a pipe. The simulator runs only while TkGate is in simulation mode and is shut down when TkGate returns to "edit" mode.

3 Virtual Peripheral Device Examples

The TkGate distribution includes two VPDs as examples to both show what can be done with VPDs and as an aid to designers of additional VPDs.

3.1 TTY

The VPD for the TTY device, shown in Figure 2, is an xterm-like window that interacts with a simulation of a small microprocessor circuit included as an example circuit with TkGate. The microprocessor is configured to run a small animal guessing game similar to "20 Questions". The VPD is implemented using the Tcl/Tk "`text`" widget. The "`bind`" command is used to intercept key presses and transmit the ascii code for the corresponding character to the Verilog stub module. The Verilog stub module in turn transmits character codes back to the VPD which displays characters using the "`insert`" widget command.

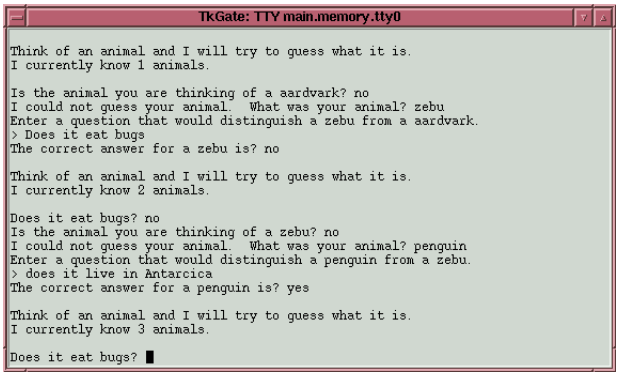

Figure 2. Window for TTY VPD

The TkGate screenshot (Figure 1) shows the TTY device as it is used in a part of the microprocessor example (located near the bottom of the screenshot). The symbol itself was created using a built-in bitmap editor that allows TkGate users to define custom symbols for new devices. The program stored in the simulated microprocessor accesses the TTY by reading and writing the memory addresses "0x10" and "0x11" as set by the TTYSTATUS and TTYDATA dip switches.

3.2 Drink Vending Machine

The drink vending machine VPD shown in Figure 3 is comprised of an external view (left side) and an internal view (right side). The external view includes buttons that can be pressed to make a drink selection, a coin slot, and a bill reader. Coins and bills are inserted by dragging and dropping from the set of coins and bills (and fake bills to test the bill reader) at the left hand side of the display. The internal view shows the columns of drinks that are available in the machine, the status of the bill scanner, the coins that have been inserted (but not used for a purchase), the coins that have been committed to a purchase, and coins that are available to make change.

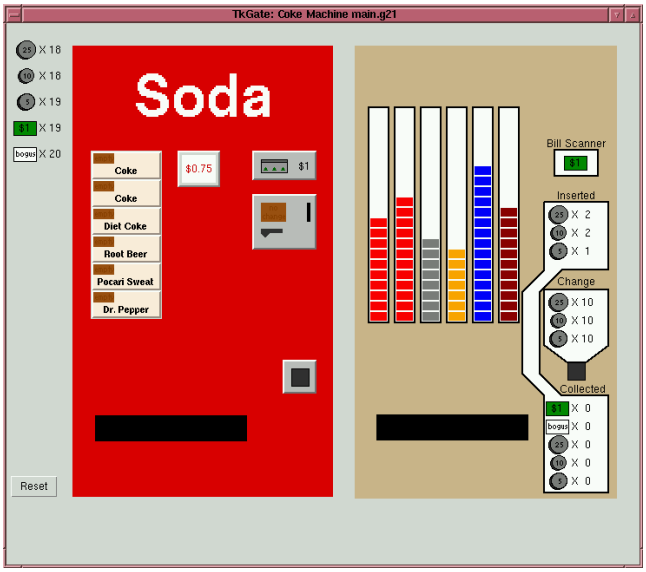

Figure 3. Window for Drink Vending Machine VPD

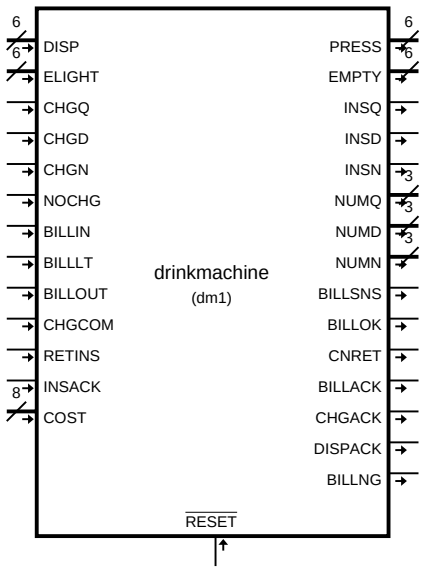

Figure 4. Drink Vending Machine Interface

User actions made through the drink machine are translated into signals on the interface of drink machine shown in Figure 4. For example, when the user pushes a drink select button, the corresponding bit of the "`PRESS`" output is asserted. Likewise input signals from the user circuit are translated into actions shown in the VPD window. For example, on the rising edge of a bit in the "`DISP`" signal, a drink from the corresponding drink column is dispensed.

4 Creating Virtual Peripheral Devices

Virtual Peripheral Devices are comprised of a Tcl/Tk script file, and a Verilog stub module. The script file implements the graphical interface for the device and handles user interaction. The Verilog stub module encapsulates the behavior of the device into a module that can be included in user circuits. The VPD script files are read both from a read-only directory that is part of the TkGate installation, and from a user-defined list of directories specified in the user's TkGate preferences file. Currently these script files are read at start-up time, but dynamic loading of VPD scripts could be easily implemented.

Since a Verilog description can contain multiple instances of a VPD, the Tcl script must be written in such a way so as to allow multiple instances of a VPD interface. This is done by giving each instance and instance name. The VPD instance name is typically the same as the Verilog instance name of the stub module for the VPD. The fully instantiated Verilog instance name is dot separated path such as "`top.bus1.dm1`".

4.1 Named Channels

Communication between the Tcl-side and Verilog-side of the VPD implementation is performed through a Verga extension to Verilog called a "named channel". A named channel is basically a queue that has a string identifier. TkGate provides a Tcl-side and a Verilog-side API to access the named channels allowing data to be passed through the channel. Named channels can be used both to send data from the Tcl side to the Verilog side, and to send data from the Verilog side to the Tcl side.

4.2 Direct Execution of Tcl Commands

It is also possible for the Verilog stub module to execute Tcl commands directly using the `$tkg$exec()` system task provided by Verga. However, use of the `$tkg$exec()` is restricted due to the fact that allowing arbitrary Tcl commands implies allowing arbitrary

shell commands. This means that untrusted circuit files could result in damage to the user's system when simulated. For this reason, TkGate provides the capability of choosing a security policy to control the use this system task. These policies are:

low Usage of `$tkg$exec()` is unrestricted. Any Tcl command, including the Tcl "`exec`" command is allowed.

medium Only registered Tcl commands may be executed. Furthermore, the "[" and "]" characters are disallowed to prevent using them to execute forbidden Tcl commands. Each VPD script file will typically register any commands it wishes to make available to `$tkg$exec()`.

high The `$tkg$exec()` task is disabled. Any VPDs that depend on using `$tkg$exec()` will not function when this security policy is selected.

Because of the potential security issues and the fact the a user could choose to use a "high" security policy, it is generally recommended that VPD implementors should avoid use of `$tkg$exec()` and use only named channels when possible.

4.3 Tcl-Side Interface

The Tcl script for a VPD is responsible for creating a window for the device, responding to user input, and communicating with the Verilog stub module. TkGate provides the following Tcl-side API for creating VPDs:

`VPD::register` *name*
: Register a new VPD named *name*. Registering a VPD allows it to be posted using the Verilog `$tkg$post()` task.

`VPD::allow` *names...*
: Register Tcl commands that can be executed from the Verilog simulation when running TkGate with medium or lower security. The '*' character can be used as a wildcard.

`VPD::disallow` *names...*
: Register Tcl commands for which execution from the Verilog simulation is explicitly disallowed when running TkGate with medium or higher security. The '*' character can be used as a wildcard.

`VPD::isallowed` *name*
: Test a procedure name to see if it can be executed from the Verilog simulation.

`VPD::shutdownnotify` *script*
: Register a script to be executed when TkGate exits simulation mode. The registration is deleted after executing the script.

`VPD::newtoplevel` *options*
: Create a top-level window that can be used for a VPD and return the name of the window. The window name is automatically generated. Top-level windows created with this command are automatically `destroy`ed when TkGate exits simulation mode. A title for the window can be specified using the `-title` option, and a command to be executed when the simulator shuts down can be specified with the `-shutdowncommand` option. The shut-down command does any additional cleanup needed by the VPD besides destroying the window.

`VPD::outsignal` *chan var*
: Cause any value assigned to *var* to be sent to the simulator over the named channel *chan*. The channel name is typically formed by using the VPD instance name as a prefix and appending a local name with a dot separator. By default, values assigned to *var* are interpreted as a decimal integer, but Verilog format constants can also be assigned as well. For example, assigning a value of "8'h3f" would cause the value to be interpreted as the 8-bit hexadecimal value '3F'. The association between the channel and the variable is automatically deleted when TkGate exits simulation mode.

`VPD::insignal` *chan [options]*
: Register an action to be taken when data is available on the named channel *chan*. Channel names are chosen in the same manner as `VPD::outsignal`. One or more options are usually given with this command. The `-command` option takes a Tcl command to be executed when data is received on `chan`. The value received on the channel is appended to the command before execution. The

`-variable` option indicates a variable to be assigned. Additionally, the `-format` switch indicates the format in which data should be reported. The format is given as a Verilog style format string such as "%d" for decimal or "%h" for hexadecimal. The association between the channel and the variable is automatically deleted when TkGate exits simulation mode.

A typical VPD script will begin by executing the `VPD::register` command to register the name of the VPD. It may then optionally use the `VPD::allow` (and `VPD::disallow`) command(s) to register specific commands that are allowed to be executed from the Verilog side using the `$tkg$exec()` system task if that interface method is used.

The remainder of the VPD script should be a set of methods defined within a `namespace` using the VPD identifier as the name. Any methods needed to support the VPD should be defined in this namespace. At a minimum, each VPD is required to provide a "`post`" method. The "`post`" method should take an instance name as its first argument, and may optionally define one or more additional arguments. The "`post`" method is generally responsible for taking the following actions:

- Create the top-level window for the VPD using the `VPD::newtoplevel` command. This window will be automatically `destroy`ed when TkGate exists simulation mode. The `-shutdowncommand` switch can be used to specify a script to execute when the window is destroyed.
- Build the widgets for the VPD in the top-level window and set up handers for user actions.
- Register input and output signals using the `VPD::insignal` and `VPD::outsignal` commands.

Note that for some VPDs there may be exceptions to these rules. It is also possible to use Tcl as glue to interface the simulation to a real-world device without using a GUI. For example, one could write a VPD to give a Verilog description the ability to access to the Internet. With such a VPD, the `VPD::shutdownnotify` command can be used to register a script to execute when the simulation is terminated.

As was stated above, there can be multiple instances of a VPD. When there are multiple instances, the "`post`" method for a VPD implementation will be called multiple times, each time with a different instance name. It is important to keep state information for each instance separate. This can be done by keeping all such state information in Tcl arrays. For example, instead of keeping state information in a flat variable such as "`$state`", the information should be kept in a variable such as "`$state($name)`" where "`$name`" is the name of the instance.

Figure 5 shows an excerpt from the Tcl script for the VPD implementing the drink vending machine. At the beginning of the script, the VPD name is registered at Line (2). The remainder of the script is defined within a namespace starting at Line (5). Variables for the top-level window as well as any other variables are declared at the top of this namespace block. The "`post`" method is defined starting at Line (16). It is passed a single argument, "`name`", that will be set to the instance name of the VPD. It begins by invoking the `VPM::newtoplevel` command to create the top-level window for the vending machine. The `-title` switch is used to set the name on the title bar of the window, and the `-shutdowncommand` switch is used to register a method to be called when TkGate exits simulation mode.

After additional Tcl commands used to configure the window, the `VPD::outsignal` command is used at Line (28) to specify that when the Tcl variable "`DrinkMachine::osigPRESS($name)`" is set, the value of that variable should be sent to the Verilog simulation on the named channel "`$name.PRESS`". The channel name is formed by appending a local name, "`PRESS`", onto the instance name of the VPD to ensure each instance has its own channel. Additional output signals can be defined in the same way.

Input signals are defined using the `VPD::insignal` command as shown on Line (29). In this case, the command indicates that when data is available on the named channel "`$name.NOCHG`", the Tcl variable "`DrinkMachine::noChange($name)`" is set to that value. The "`-format %d`" switch indicates the format to use when assigning a value to the variable. Format conversion is performed by the simulator, and the format should be a specifier supported by the Verilog `$display()` task. It is also possible to use the `-command` switch to specify a script to execute when data is received. One or both of the `-variable` and `-command` switches can be specified.

```
(1)   # Register the name of this VPD
(2)   VPD::register DrinkMachine
(3)   ...
(4)
(5)   namespace eval DrinkMachine {
(6)     # Array (indexed by instance name) for the top-level window
(7)     variable vm_w
(8)
(9)     # Array (indexed by instance name) for the ``no change'' light.
(10)    variable noChange
(11)
(12)    # Array (indexed by instance name) with the state of the drink select buttons.
(13)    variable osigPRESS
(14)    ...
(15)
(16)    proc post {name} {
(17)      # Namespace variables used by post method
(18)      variable vm_w
(19)      variable noChange
(20)      variable osigPRESS
(21)
(22)      # Create top-level window for drink machine
(23)      set vm_w($name) [VPD::newtoplevel -title "Vending Machine $name" \
(24)         -shutdowncommand "DrinkMachine::unpost $name"]
(25)      ...
(26)
(27)      # Register named channel with button state variable
(28)      VPD::outsignal $name.PRESS DrinkMachine::osigPRESS($name)
(29)      VPD::insignal $name.NOCHG -variable DrinkMachine::noChange($name) -format %d
(30)      ...
(31)    }
(32)  }
```

Figure 5. Tcl-Side Interface to Vending Machine VPD

4.4 Verilog-Side Interface

The purpose of the Verilog stub module for a VPD is to encapsulate the channel I/O operations between the Tcl side and the Verilog side into a module that can be included and used like as a regular Verilog module within a user circuit. The stub module is usually defined in a library that is included by user's circuit. The Verga Verilog simulator supports the following API system tasks in support of VPDs:

`$tkg$exec(`*format*`,` $p_1, \ldots, p_n$`)`

Constructs a string for a Tcl command and sends an execution request from the simulation to the main TkGate executable. The string is constructed similar to the Verilog `$display()` task (which is in turn similar to the C `printf()` function). The Tcl command is executed asynchronously with `$tkg$exec()` not waiting for the command to complete. A "`%m`" in the "*format*" string will be substituted with the name of the current instance. This is typically used as the instance name of the VPD.

```
(1)   module drinkmachine(..., PRESS, NOCHG, ...);
(2)   output [5:0] PRESS;
(3)   reg [5:0] PRESS;
(4)   input NOCHG;
(5)
(6)     //
(7)     // Execute the drink machine post command to start up the Tcl/Tk interface.
(8)     //
(9)     initial $tkg$post("DrinkMachine","%m");
(10)
(11)    ...
(12)
(13)    //
(14)    // Respond to changes in the Tcl/Tk osigPRESS variable.
(15)    //
(16)    always #10 PRESS = $tkg$recv("%m.PRESS");
(17)
(18)    //
(19)    // Send updated value of NOCHG signal to Tcl/Tk side of VPD.
(20)    //
(21)    always @ (NOCHG) $tkg$send("%m.NOCHG",NOCHG);
(22)
(23)  endmodule
```

Figure 6. Verilog-Side Interface to VPD

`$tkg$post(`*vpdname*`,`*instname*`,` $p_1, \ldots, p_n$`)`

Post an instance of the VPD named "`vpdname`". The effect is similar to executing "`$tkg$exec("`*vpdname*`::post` *instname* $p_1 \ldots p_n$`")`" except that it is not subject to the security restrictions of `$tkg$exec()` as long as *vpdname* is a registered VPD. Any "[" and "]" characters are treated as ordinary characters and each p_i is passed as a single parameter to the `post` method even if it contains spaces. This task also executes the Tcl command asynchronously and does not wait for it to complete. A "`%m`" in any parameter will be substituted with the name of the current instance. In most cases, "`%m`" should be passed as the *instname*.

`$tkg$send(`*name*`,` *data*`)`

Send *data* on the named channel *name*. If being used to send data to the Tcl side of a VPD, the channel name should correspond to a channel name specified in a `VPD::insignal` command. The transmitted data will cause either a Tcl variable to be set or a Tcl procedure to be invoked. A "`%m`" can be used in the channel name will be substituted with the name of the current instance. This can be used to create a compound name such as "`%m.NOCHG`".

`$tkg$recv(`*name*`)`

Returns data received on the named channel *name*. If being used to receive data from the Tcl side of a VPD, the channel name should correspond to a channel name specified in a `VPD::outsignal` command. When the variable declared in the `VPD::outsignal` command is set, the value of that data, interpreted as a decimal value, will be available to be read by this task. The task will block if there is no data in the channel. A "`%m`" can be used in the channel name will be substituted with the name of the current instance. This can be used to create a compound name such as "`%m.PRESS`".

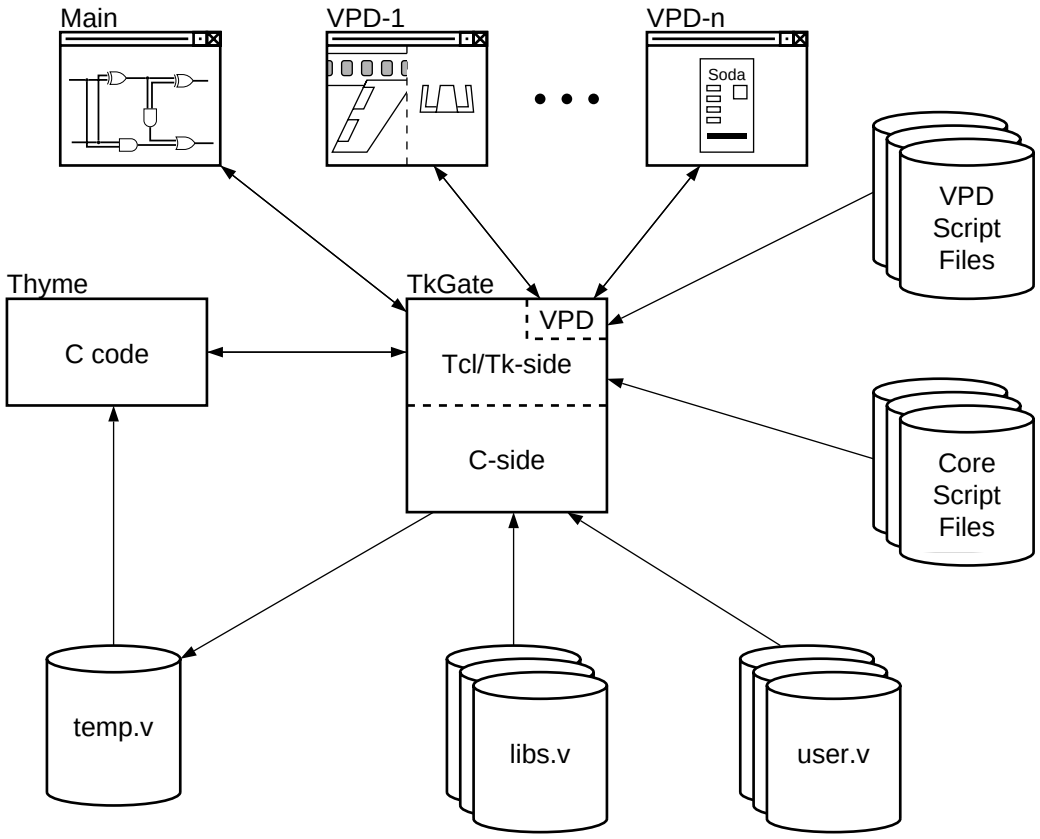

Figure 8. TkGate/VPD Architecture

```
module top;
  wire [5:0] PRESS1, PRESS2;
  wire NOCHG1, NOCHG2;
  ...
  drinkmachine dm1(..., PRESS1, NOCHG1, ...);
  drinkmachine dm2(..., PRESS2, NOCHG2, ...);
  ...
endmodule
```

Figure 7. Top-Level Circuit Using Two Vending Machine Instances

A typical Verilog module for a VPD will invoke the `$tkg$post()` task in a Verilog "`initial`" block. In a Verilog, an "`initial`" block is used to specify Verilog code that should be executed once at the start of a simulation. All of the "`initial`" threads are begun in parallel with respect to simulation time, that is, the time in the circuit that is being simulated. Placing the call to `$tkg$post()` here causes all instances of all VPDs to create their windows as soon as the simulation is started.

Figure 6 shows the Verilog stub for the drink vending example. Line (9) causes the `$tkg$post()` command to be executed as soon as the simulation begins. This task call will in turn cause the Tcl procedure `DrinkMachine::post` to be invoked.

Signals input from the Tcl side of the VPD can be handled using a Verilog "`always`" block as shown on Line (16). An "`always`" block is essentially an infinite loop executed in its own thread. The " #10" in this example indicates a delay of 10 time units after which the `$tkg$recv` task will be executed to read data on the named channel "`%m.PRESS`. The data is then placed into the Verilog variable `PRESS` which is declared as a register and is also an output of the module.

Signals to be output to the Tcl side of the VPD can be handled using an "`always`" block as shown on Line (21). The Verilog "`@(NOCHG)`" construct blocks until

the value of NOCHG changes. As soon as NOCHG is given a new value within the simulation, the tkgsend() task is invoked to send the data to the Tcl side.

Implementation of the Verilog stub is not limited to these two types of constructs. For example, a VPD implementing a TTY device might have a transmit data register TD which is sent to the Tcl interface when a rising edge occurs on a separate transmit signal TX. This behavior might be encoded using the Verilog construct:

```
always @(posedge TX)
  $tkg$send("%m.DATA",TD);
```

More complex protocols are also possible. For example one may wish to allow transmission only when a clear to send signal from the VPD device is asserted, and require an acknowledge before transmitting another character.

Once the stub module for a VPD has been written, it can then be included in a client Verilog description. For example, Figure 7 shows a top-level Verilog module that uses two instances of the drink vending machine VPD. When this Verilog description is simulated, two drink machine windows will be created with the titles "Vending Machine top.dm1" and "Vending Machine top.dm2" (see Line (23) in Figure 5). The names of the instances are "top.dm1" and "top.dm2". Each instance can be connected to a separate set of input and output signals.

5 Implementation

A block diagram of TkGate is shown in Figure 8. TkGate itself consists of a Tcl/Tk portion and a C portion. On startup, the core TkGate scripts as well as the VPD script files are read. The user circuit is read by the C-side of TkGate from one or more circuit files and any required library files including library files containing VPD stub modules. When starting a simulation, TkGate writes all of the currently defined modules to a temporary Verilog file which is passed to Verga for simulation. As the simulation begins, the *tkg*post() task is called for each each instance of a VPD in the circuit. This causes commands to be sent back to the TkGate interface which cause the "post" method to be called for each of these VPD instances. This in turn results in TkGate creating a window for each active VPD.

The simulator is invoked from the interface using the Tcl "open" command to create a bidirectional pipe. The pipe is used to send commands between the interface and the simulator. Both the simulator and the interface have a private set of commands that they recognize to control the simulation and report results. Normally these commands are hidden from the user. TkGate uses the Tcl "fileevent" command to register a function to be called when commands are ready to be read. This allows it to handle interactions with the interface while handling commands from the simulator as they arrive.

We will illustrate what happens internally through an example. The following illustrates the steps that are taken when starting a simulation with a VPD from TkGate:

1. TkGate determines the modules that are required by the simulator, writes a Verilog file in /tmp, and invokes the simulator using "open" with a mode of "r+" .

2. TkGate sets the variable Simulator::isActive to "1" to indicate that the simulator is active.

3. The Verilog stub module for the VPD executes "tkgpost("DrinkMachine","%m")" as the simulation begins.

4. The simulator sends the command "post DrinkMachine top.dm1" to TkGate.

5. TkGate uses Tcl_ExecObjv() to execute the Tcl command "DrinkMachine::post top.dm1".

6. The DrinkMachine::post method calls VPD::newtoplevel to create a top-level window for the VPD.

7. VPD::newtoplevel creates the window and uses "trace add variable" on Simulator::isActive to detect when the simulation is terminated.

After building the interface widgets, the DrinkMachine::post method may make calls to VPD::outsignal to register output signals. The VPD::outsignal uses the Tcl "trace add variable" command to set a trace on the registered variable so as to call the private method VPD::sendVariable whenever the variable changes. For example, the VPD::outsignal command on Line (28) of Figure 5 would result in the Verga control command "$write top.dm1.PRESS 4"

being sent to the simulator when the variable "DrinkMachine::osigPRESS(top.dm1.PRESS)" was set to "4". The VPD::outsignal command also sets a trace on Simulator::isActive so as to delete the signal registration when the simulation is terminated.

In addition, the DrinkMachine::post method may also make calls to VPD::insignal to set a handler for incoming data on a named channel. This is done by sending a message to the simulator to watch a specific queue and by recording the actions to take place when data is received on that queue. For example, the VPD::insignal command on Line (29) of Figure 5 would result in the Verga control command "$watch $queue top.dm1.NOCHG %d" being sent to the simulator over the pipe between TkGate and Verga (the last argument is the format to use). Verga will then send a command such as "tell queue top.dm1.NOCHG 1" over this pipe when a thread in the Verilog description executes the tkgwrite() task on the channel "top.dm1.NOCHG". In addition, any pending values on the channel are sent immediately on receipt of the "$watch $queue" control command.

When the user terminates the simulation, TkGate sends the control command "$exit" to Verga. Verga then replies with its own "exit" command to TkGate and calls "exit(0)" to terminate itself. When TkGate receives the "exit" command from Verga, it closes the pipe and sets the "Simulation::isActive" flag to 0. This in turns causes VPD windows created through VPD::newtoplevel to be destroyed using the "destroy" command. It also causes any registered shutdown function to be executed and any variable traces set by VPD::outsignal to be deleted.

6 Conclusion

In this paper we have presented the Virtual Peripheral Devices (VPD) feature of TkGate. We showed examples of two existing VPDs, showed how to create VPDs on both the Tcl and Verilog side, and described TkGate's implementation of the VPD feature.

References

[1] Jeffery P. Hansen, "TkGate: A Schematic Capture and Digital Circuit Simulation Tool", http://www.tkgate.org, March 2000

[2] Samir Palnitkar, "Verilog Hdl : A Guide to Digital Design and Synthesis (2nd ED)", SunSoft Press/Prentice Hall, 2003