Using Tcl /CCI to Turn EDA Cousins into Sisters

Dwight Hill
Synopsys

Abstract

The Tcl language, augmented with the Synopsys Common Command Interpreter (CCI) and *collections*, is now the primary user interface for a suite of Design Implementation tools, each of which runs on a database optimized for its own purpose. This system provides efficient access to design and library data, while retaining flexibility in the architecture of the underlaying databases. Its success demonstrates that portability of interface is more important than portability of database.

Background and Motivation

All practical RTL-to-GDSII flows involve multiple tools supporting such diverse activities as simulation, synthesis, placement, routing, chip finishing and signoff timing. Whether these tools come from the same vendor or multiple vendors, they are each optimized to perform a specific function. High performance implies not only efficient algorithms, but representations for designs and libraries that are tuned to each application.

While such multiple representations are key to good tool performance, if they are exposed to designers they can add overhead and lead to increased start up time, frustration, and overall time to results. Multiple interfaces can also make it difficult to reuse training materials or command scripts across tools.

One approach to this is to design an all encompassing data model that can represent the design and its libraries and other ancillary information from RTL through GDSII. Such an approach works by compromising the access for some operations (such as logic synthesis) in order to support other operations (such as slotting). Even if the system is designed from scratch, spanning such a wide spectrum can be an exercise in “robbing Peter to pay Paul”. Getting this to actually work over time can be problematic because the requirements for each level of design are constantly evolving, and new features must be integrated across a wide spectrum of views. But even when it works, it is unlikely to remain competitive with data models tuned to specific facets of the problem.

Requirements for an Interface

In order to be effective, an EDA interface must support observability and control and must do so efficiently and conveniently. That is, the interface must allow its users access to all the relevant information about the design and its environment (*e.g.* libraries, technologies, etc.). And it must allow all aspects of the design that can be modified to be modified using just the interface commands without resorting to external mechanisms such as hand editing, special GUI tools, or proprietary database packages. Conversely, since some information is not intended to be modified during a session, the interface should not enable modification of this data (*e.g.* timing libraries).

The effectiveness and efficiency of the interface needs to be such that the interface itself does not form a high percentage of the run-time or memory usage of the tool. For sake of concreteness, one might place this at 10%. So the interface to a router, for example, that runs 10 hours on a million gate design, should not require more than 1 hour just supporting the interface, nor more than 10% of the memory or disk resources. If the router runs only 10 minutes, the interface should take less than one minute. Generally, interactive support is the most demanding. A graphical

editor, for example, might respond to a mouse click command to move a group of wires in less than 1 second. By this rule, the interface portion of the time should be less than 100 milliseconds. If the user embeds hundreds of commands into a script, and sources the script, the overall time per command is generally much less than 1 second. Thus, it makes sense to tune the interface to support its most demanding application: i.e. interactive/ scripted work. If the interface works well enough to support these it will probably be more than sufficient for other, more batch oriented applications.

Solution Description

The solution described here is basically to use Tcl, with CCI and collections to present a unified interface to the multiple environments. Used in this way Tcl provides not just a user interface but actually an application interface, that human users and even some locally generated tools can build upon. Even GUI architectures can be built upon the Tcl/CCI/Collections layer with minimal overhead.

Background on CCI and Collections

The architecture of CCI involves enhancing standard Tcl, not modifying it. During input, CCI acts as a layer between the text typed in by a user and the standard Tcl interpreter. As the command is executed, CCI works between the tokens scanned by Tcl and the interpretation of those commands, to simplify the job of the C programmer who is implementing commands. An example of the first type of operation is the interception of array notation. In an ordinary Tcl shell, the square brackets "[]" denote the execution of a command inside a command. Thus the command

```
set x bus[3]
```

would trigger an error in ordinary Tcl because the token "3" is not an executable command. But CCI preprocesses this command to make the string "bus[3]" to be a token. This is critical because EDA objects often have brackets in them. Note that the [command] syntax is still supported for actually invoking sub commands when needed.

An example of the second type of operation CCI provides would be processing a command like

```
set_false_pat -fro [get_pins abc/d*]
```

In this case, CCI would understand that "set_false_pat" was an abbreviation of "set_false_path", and -fro was an abbreviation of "-from". CCI would also handle the fact that the command actually has more than 12 non-required, named options: they could be specified in any order, and their interdependencies checked. For example, some options are mutually exclusive, have a range bound, or must be used with one of a specific enumerated set of tokens such as "low", "medium" or "high". The developer of commands need not check these conditions in his code, because commands that violate these syntax pre-requisites are blocked by the CCI layer.

Other Tcl usability extensions include uniform help facilities, which are derived directly from the source code and are so guaranteed to be up to date, global Tcl variables with specific types and type checking, and more. For complete documentation on CCI, the reader should consult the *Primetime User Guide: Advanced Timing Analysis* manual.

Collections

Collection operations usually center around "get_" commands, as in

```
set inputs [get_ports -filter
"direction==input" J*7]
```

This command finds all the ports that match the expression "J*7", which might include "Jack77" or "J_7x7", *etc*., and then filters them down by direction. In this case ports that are not of direction "input" are excluded. The remaining ports, if any, are stored in the Tcl variable "inputs", as a collection.

Collections are efficient internal representations of design objects: they are not just strings or Tcl lists. They describe the class of the objects that comprise

them: a collection with a port NN is completely distinct from a collection with a cell NN. The underlying implementation of collections can be tuned to the database that the tool operates on. If it is an in-memory representation, the tool can use ephemeral (C or C++) language objects. If the database has a persistent component, collections can be implemented using on-disk reference mechanism. But the user need not be concerned with this. To him or her, the semantics of collections are defined by the Tcl commands that create, modify and accept them. Removing collections is simple: as soon as the Tcl variable that is the handle of the collection is no longer accessible, the collection is freed.

The Tcl/CCI/collection syntax was popular enough that it has formed the basis of the Synopsys Design Constraint (SDC) language, which is now a widely accepted format for expressing design constraints. Its success demonstrates that portability of interface is more important than portability of database. However, SDC uses only a fraction of the power of Tcl/CCI/Collections, because most tools do not support the broad range of operations implied by CCI. These tools can process only timing constraints (not general commands), and cannot control the set of objects processed or extend themselves to new commands in a compatible manner.

Non collectable design information

Note that not every design quantity is a first-class object. For example, a layer known to a router is not collectable object, nor are operating conditions (for timing), nor in general points, lines, or rectilinear shapes. However, Tcl can easily express these concepts using it's built in types which include integers, strings, lists of strings. To make these useful, however, the tools involved must agree on the syntax and semantics of these Tcl constructs. For example, the syntax for a distance is generally a fixed point number; the syntax for a point is a Tcl list with two elements, the first being the X coordinate and the second being the Y coordinate. These syntax and semantics can be enforced and leveraged with C and Tcl libraries that generate, validate, and accept parameters in this format.

Accessing these data can be accomplished through Tcl commands returning the appropriate values. Probably the most common command is "get_attribute", which finds a particular attribute on a first class object. This attribute may be a string, such as its name, a list, such as its bounding box, or a collection, such as the set of nets that interact with it for signal integrity calculations. Tcl commands that return strings are convenient to use directly, because the result is printed directly at the terminal, and also convenient to use inside of Tcl procs where the results can be gathered up into Tcl variables and processed into a more useful format. **N.B**. Attributes on first class objects are not necessarily correlated with the datamodel structures. Unlike "common database" architectures, tools in a suite united by Tcl may generally attributes that are computed on the fly, rather than defined by a schema. Typical examples of these are the "area" attribute, which is computed by multiplying the difference in the corner coordinates, or the "wirelength" attribute, which is computed by traversing a potentially complex topology.

Putting Tcl/CCI/Collections onto a suite of tools

The most important decision when designing the interface provided by Tcl/CCI/Collections is determining the set of "first-class" objects that the tools will support. The names, attributes and semantics of these objects will determine much of the rest of the user interface. Generally, first-class objects are those that:

- may appear in collections
- have a name (although in some cases, this is not relevant or even directly ac-

cessible)

- can be obtained using a get_<class> command, such as “get_pins”
- may have attributes. All object classes have some attributes built-into the tool (such as “name” and “object_class”), but the tool will generally also support user-defined attributes.

Obviously, the first set of objects that need to be implemented in a tool will include netlist items, such as nets, ports, and pins. Defining their semantics, however, is not always obvious. When tools come from distinct backgrounds, or work with users have differing expectations, the definition of “port” and “pin”, for example, may have been reversed in their original user presentation. By building a Tcl/CCI environment into the tools, the new interface has a chance to unify the concept. Internally, the tools may have different databases, such as Synopsys db, Milkyway, or Open Access, and these may have different “5-box” models for net lists, but when working through the Tcl interface they will appear to be equivalent. This can be accomplished even if the underlying representations are structurally disjoint. For example, the expression

```
get_cells a/b/c/d/e
```

may refer to a cell called “e” that is four levels deep in a hierarchy, with a parent named “d”. In some situations, it might refer to a cell called “a/b/c/d/e”, which is at the topmost (and only) level of the hierarchy. In general, placement and routing tools tend to have flatter hierarchies than logic synthesis or simulation tools that work with a deep logical hierarchy. But by agreeing on a common syntax and semantics for Tcl/CCI these tools can communicate efficiently and conveniently.

Internal Architecture Options

One of the advantages of a common Tcl interface is that it can be implemented in a variety of ways to leverage the strengths, and avoid the problems that may exist in a tool. The most common way to implement a Tcl /CCI command is to write a C (or C++) function that manipulates the design, and register the function with the CCI system. The command function can access data from the native tool representation, or even from other representations as needed. In fact it is even possible to have one representation create a collection and another command, written to accept another representation accept it by using a collection translation mechanism. For example, a command like “all_fanouts” might be conveniently implemented in a logical or timing environment. (This command returns a collection of pins that are in the logical fanout of a specified pin). A separate command, perhaps a placement or routing command (such as rip_up_route) could accept this collection even if it were implemented on a different representation. This translation function can be accomplished internally to the Collection system without programmer intervention (or even the knowledge that translation was taking place).

Because Tcl an extensible language it is possible to define new commands using preexisting Tcl commands or even commands in other languages, as needed. These can be integrated into the tool in such a way as to appear to be native commands, using “define_proc_attributes” so that the user procs are indestinguishable from vendor commands, with consistent options processing, help and documentation. Note that the set of potential benefactors of this includes CAE’s, field engineers, corporate CAD groups, *etc.* and is in the thousands. By contrast, only a relatively small number of C/C++ programmers work with a common data framework.

Objective: Sharing Scriptware

Any engineer who has used EDA software for more than about 15 minutes reads, writes and depends on scripts to make his or her flow work. If several tools use the same database, but have disparate interfaces, it may be OK to move a design from one tool to another, but it will be impossible to transfer a script written in one to another. But when the tools are united under Tcl/CCI it becomes feasible to share scripts. The wide acceptance of

SDC is evidence of the success of this: SDC has been implemented in many tools, and many databases, from many companies.

Still greater integration is possible if the tools are intentionally architected to use the same Tcl interface. Rather than just having each tool define commands independently, it becomes possible for the commands to be pre-analyzed for compatibility (or at least non-interference.) Specifically, Tcl/CCI classes can be grouped into several categories:

- Objects common to all tools in the suite, such as nets, ports, etc. These generally must be supported in compatible manner through the whole flow.
- Objects local to one tool (not present in any common database), such as slots in a slot-filling step.
- Objects common to two or more tools in the suite communicating through databases or translators, but not necessarily understood by all other tools. For example, physical implementation tools, such as routers and DRC engines will need “wire” objects, but logic simulation tools will generally not need them.

The second most important thing in architecting these interfaces is that the same concept have the same name in all tools that use it. The Most Important Factor is that different concepts have different names in all tools. It would be major productivity hit if one tool considers a “bound” to be a a move bound (for keeping cells inside), and another tool considers it a limit on the power that a block may dissipate. It is almost as much of a disaster for EDA developers as for users. But the use of a common Tcl/CCI/Collection interface can avoid this situation.

In addition to defining the object types consistently, object attributes names, and more importantly, semantics, should be coordinated. For example, the “bounding_box” attribute of all objects needs to be defined as consisting of a Tcl list comprised of one point (the lower-left corner) followed by another point (the upper-right corner), in all tools that support it. If the tools use the same database, but choose to describe bounding boxes as four integers, it will be impossible to share scripts, or even training, across the tools.

Finally, command names need to be consistent. The SDC standard is commands with all lower case letters, with a verb_[modifier]_noun structure, as in get_cells, set_false_path, report_ports, or remove_objects. Since many of the <noun> clauses will follow the object class names selected, careful object class name selection will get you halfway to consistent command selection. The <verb> portion needs to be consistent across tools as much as possible, but it is generally OK to have each tool in the suite to have some unique verbs. After all, that is the tool’s *raison d’être*.

Tcl as the Controlling Interface For Large Engines

Large complicated engines, such as timing analyzers or routers, have large complicated interfaces. Getting straightforward information like a netlist in is typically not an architecture problem. But controlling and accessing all of the engine’s facilities is. Large engines often support hundreds of operations, each with many options, and controlled by dozens of environment settings. When another system, such as a time budgeter, needs to interface with these large engines, the system needs access to essentially all the “knobs and meters” provided as part of the engine’s user interface. The classic way to provide this is to have one C function (or C++ method) to set or get each datum that would be controllable by the user interface. But this essentially doubles the size of the interface. Each facility must be supported in both a C/C++ language and user-interface language format. [These API’s can get quite large. One standard has more than 9Mbytes just of API.]

A more effective solution is to have the engine support a Tcl interface for the bulk of operations. The calling system can then assemble commands as strings, and have them evaluated by the engine. Likewise, it can retreive data back though the return values of Tcl commands. Moreover, bugs can be tracked down much more quickly and easily: the sequence of commands essentially comprises a bug report that is human readable, and often, replayable.

Developing tools at this level provides a higher level of efficiency than the older style “object” or API based systems. There is no need to worry about memory management or garbage collection, design object typing is automatic, attributes can be extended without effecting global header files. All of these let the user focus on the function without getting stuck with lots of low-level C/C++ language issues. Perhaps equally importantly, tools developed in Tcl/CCI can ride out upgrades or even replacements in the data model much more easily, as long as the new model can support the same Tcl commands. Many parts of the tool will not even require recompilation.

Efficiency

In order to be effective the Tcl/CCI/Collections layer must not only be convenient for humans entering commands at a rate of one per second, but efficient enough to process thousands, or even tens of thousands of operations per second. Without this speed the interface would not really support scripting, since scripts, including SDC, tend to have sizes proportional to the size of the design. In the DSM era million-line SDC scripts are common.

Fortunately, the Tcl/CCI processor itself is very fast. With simple commands, its speed is almost limited by the ability to get large blocks of text over the network and into the Tcl parser. The collection mechanism is also efficient in two dimensions (at least). First, collections allow large groups of objects to be gathered together with minimum memory or CPU overhead independently of their names. A system that depends on storing the names of objects, by contrast, tends to bog down in large designs because these tend to have long names constructed by flattening of hierarchy. The other dimension of performance is the ability to access individual, or small groups of objects by name. This is typically important for timing constraints because timing constraints tend to be specified on individual objects, but there are so many constraints that each of them must be accessed quickly. Again, the collection mechanism has proven to be a portable, and effective way of dealing with many independent objects.

Summary

The Tcl based interface system described here represents an level of integration less tightly constrained (from a developer’s perspective) than the sharing of a common database, but more coordinated, from a user’s perspective, than multiple tools. Indeed, the distinction between multiple executables with compatible interfaces, and a single executable may be less than between different parts of a single large program. In several situations, particularly in timing, the use of Tcl/CCI/collections has provided an efficient interface across not just multiple engines, but even across multiple companies, including Synopsys, Cadence, Magma and others.

Acknowledgements

The CCI/Collection system was designed and developed primarily by the CLT/Primetime group in the mid 1990’s. This group included Andy Siegel, Bill Mullen and others who contributed to the design and coding. Also, thanks Ulick Malone, Jeff Loescher, and Noel Strader who reviewed and enhanced this manuscript.