# An Efficient Method for Rendering Design Schematics Using Tcl/Tk, and Distributed Relational Databases.

Manu Goel(manu_goel@mentor.com), Antara Ghosh(Antara_ghosh@mentor.com)
Mentor Graphics Corporation

**Abstract**:

Debugging a design in EDA is always a challenging and time consuming process. Designers need to have access to an efficient tool which can provide them the design connectivity in a logical and efficient manner. This paper discusses various challenges faced while writing such a tool for debugging a design and how were they handled to provide a fast and efficient solution. Schematic browser is a Tcl/Tk based GUI application, which user can use interactively to debug and understand the design.

**Glossary**:

Description of terms used in the paper:

Schematic Browser – Widget to view/trace the RTL level connectivity of a signal in a design
Incremental Mode – Browse the connectivity incrementally based on need
Full View Mode – View the connectivity of a particular segment of design in one go
Waveform Viewer – Widget to view the signal waveforms

**Introduction:**

The widget being discussed here is the schematic widget, which is a part of GUI provided with a typical simulator. The GUI is used to run simulation, view waveforms and then debug user design in case of any issues, using Schematic window/Wave window. Typically, user needs to compile the design with flags to turn on debugging and then use the GUI to verify/debug the design.

Schematic browser shows a graphical representation of a user's design. The tool converts the RTL constructs of user design, to their equivalent graphical symbols and presents them. The tool should be easy to use interactively to debug or understand the design. Viewing the design graphically, results into much faster debugging and clear insight in the design, making it easy for the user to correlate quickly about how his chip is going to behave.

Schematic window has two modes –

1. Interactive Design Mode: The purpose of this mode is to debug the design incrementally. For example, if the user finds any mismatch in his design output, they will start tracing the design in schematic window starting with the first signal which shows mismatch. User can then select any net and can choose to see the drivers or readers of the selected net to see the connectivity between various constructs around the net of interest. This mechanism can help him in identifying any misconnection or any unconnected logic.

2. Full Design mode: This mode is used to create full understanding of the design and it shows one full module at a time. This mode provides a compact view initially. User can expand to see more details on their area of interest and can compact that again whenever needed.

3. Batch Mode: There is a mode in this tool, where user can use the tool even without bringing up the GUI. This is called batch mode. In this mode, user gets an interactive prompt at the terminal itself, where user can perform certain operations. User can still use some of the above mentioned features in this mode. For example, they can request certain details through command and the details will be provided in text format.
   Even when the GUI is up, user still has access to the prompt, and from there as well user can perform certain operations without actually opening up the Schematic browser GUI.

The GUI and the Schematic widget in discussion here are based on Tcl/Tk and the debug information is stored in a database software tool. User has the flexibility to open multiple instances of the Schematic browser and can perform independent operations in all of them in parallel.

**Problem Statement:**

As discussed, schematic viewer should be a intuitive tool for debugging any design. The Schematic viewer must give absolute clarity and maximum insight in the design to user, in real time. However, when the design is big, so is the netlist debug database. To manage such huge amount of data, fetch relevant information and drawing it in real time is a daunting task. To achieve that one must make sure there is as little as possible database interaction. That is, same query should not go to the database again and again.

So in one hand, the data access should be managed in a way, so that user can open debug netlist in multiple windows separately. These windows should be completely modular in behavior. Any change in one window, should not affect other windows in any manner. Consider a typical schematic rendering flow. Whenever logic is drawn in

schematic window, there are safeguards to avoid painting same logic again. The scenario of repeated rendering can occur in two cases. One case is, the design has some looped logic, and while path browsing and incrementally drawing, the tool might go through same logic repeatedly. Second case is, the user has issued command to draw same logic more than once. To avoid these, there should be information present against every window, about what logic is already present in Schematic window. These caches of information is checked before drawing any logic, so that for an already drawn logic, the whole process of data fetching, processing and drawing is not repeated. Every netlist object must be processed (processing being the cycle from data fetching to netlist rendering) only once. However the tool must make sure, if the same net is drawn in different schematic window, which should be allowed. This is needed to maintain the window functional modularity as mentioned above.

On the other hand the tool needs to keep database interaction minimum. For example, as mentioned above, same logic should not be drawn in single Schematic window more than once, but same logic can be drawn in different schematic window. However, the effort of information fetching and processing should not be repeated for same logic. This is cardinal as, multiple accesses to debug database is costly and should be strictly guarded against.

The solution for this is to keep the data pool common among different Schematic window. Database access and initial data processing, which is same for all logic, regardless of which schematic window needs the information, should be done in a way so that the effort is not repeated. This is a must for good performance.

So the system has two apparently clashing goals. One is to keep the data model as mutually exclusive as possible to have correct functionality of multiple schematic viewers. The other is to have a common data pool and data processing algorithm.

Added to this is, another use of the system is working of Batch mode. As discussed in the introduction, this mode does not need any GUI window, so the system of information caching needed for schematic windows are not needed here. However this mode can also use the common data pool.

Lastly one must understand, as design gets bigger, DB size also gets bigger impacting the performance in multiple ways –

- Loading the whole DB may take a lot of time
- If full DB is loaded in memory, then memory footprint will increase causing the system to slow down.
- Fetching the required information will be slower

So handling of database also have to be clever. Creating a monolithic database for whole design, and loading the whole database in memory, irrespective of user debug interest locality is wasteful and will harm netlist drawer’s performance.

So to sum up, for schematic to be truly useful, it must have correct functionality, it must be reentrant and fast. The performance (time and memory) is almost as important as functionality is, for schematic debugging.

**Solution**

In order to create such a tool, the basic requirements are

- Tool should support multiple windows, which can provide similar functionality, but should be completely independent
- It should provide a clear interface to database from where all the necessary information can be fetched
- Whatever information is once processed should not be processed again.
- Non-GUI mode should also work

In order to provide the above functionality, advantage of **object oriented Tcl/Tk** is taken to create the main Schematic window widget. All common functionality that has to be provided and needed to be localized to a single window can be encapsulated inside a class. This class should have functionality of both, incremental and full view mode. Information, once loaded in a window needs to be cached, so that it can be brought back very quickly if user performs the same operation in that particular window. Such information is dependent on context of the window. So a localized caching is a must for such operations. This caching data structure, resides in the class created for the window. The class will also store all the user specified preferences for that particular window.

Second part of the problem is, to fetch the necessary information from the database to show the required functionality in GUI windows, as well as in non-GUI mode. Since a lot of information may be shared among various windows, keeping the code to fetch and store the information separate is a good idea. Since this information may be needed for non-GUI mode as well, it has to be outside the scope of main class creating the widget. Apart from this, since the design connectivity information will be same irrespective of the window from where the information is being requested, all information fetched for this is cached and can cater to future requests without having to go to debug database. So **Namespace** feature of Tcl/Tk came very handy here.

All interface APIs were protected inside the name space. The caching arrays were also protected inside the name space avoiding any misuse of these caches. Further, keeping these interfaces and caching outside the main class also helps batch mode, because that is not associated to any window, so the tool does not need to create any window object for non-GUI mode. It can simply work through fetching the information directly from these name space APIs.

Since the tool uses a lot of caching and the advantage of cache can be fully achieved only if the cached information can be fetched real quickly. The **associative arrays** of

Tcl were of great help for such a purpose. The logic of interest automatically became the key for such an array to store the information in the cache, and to fetch. One simply needs to check if such an entry exists in the concerned cache or not, and if it exists then the information is available very quickly. It does not require any hashing function implementation to store or retrieve the information form cache.

The information to be cached in these arrays is of the form of
`Readers_of_net(/top/mid/in1) {/top/mid/o1 /top/mid/o2…}`

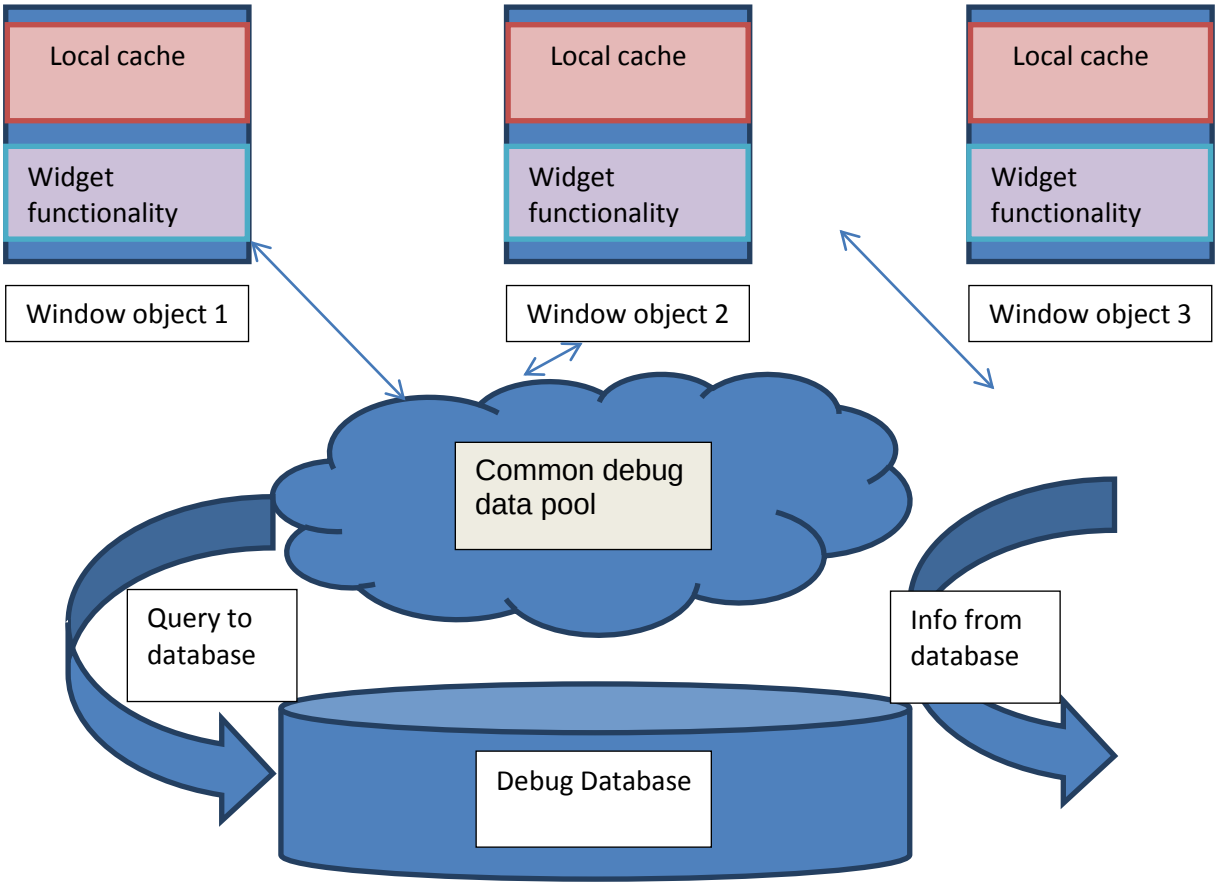


*Figure 1: Window widget and debug data pool database*

This is how the GUI side problem was solved, now comes the problem of managing the big database. Once that problem is solved, one can achieve a real efficient functionality that is needed. For this, again, Tcl came in very handy. The handling of a big database is divided into two parts. One, database should be modeled in such a way that data can be accessed efficiently. Two the data already fetched into memory should be cached and shared in judicious way among the different processes can access it independently.

The usual size of designs that simulator handle is hundreds of million gates. The debug database size can easily run into several hundred gigabytes. To keep a single monolithic database of that size and fetching information from it is time consuming. So instead of keeping single large design database, GUI should have several small databases for different parts of design.

Dividing databases into several parts also gives GUI the flexibility of generating different amount of debug information for different parts of the design, depending on the user's requirement. The simulator tool's debug database is arranged in a way that the tool maintains several databases for different parts of the design. These databases may have different amount of granularity of information on the parts of design it represents. This depth of the information for each part is dictated by the user. There is always one top node database that does the book keeping for the whole design. It would keep record of what part of design resides in which database. Also it keeps all the global information of the database.

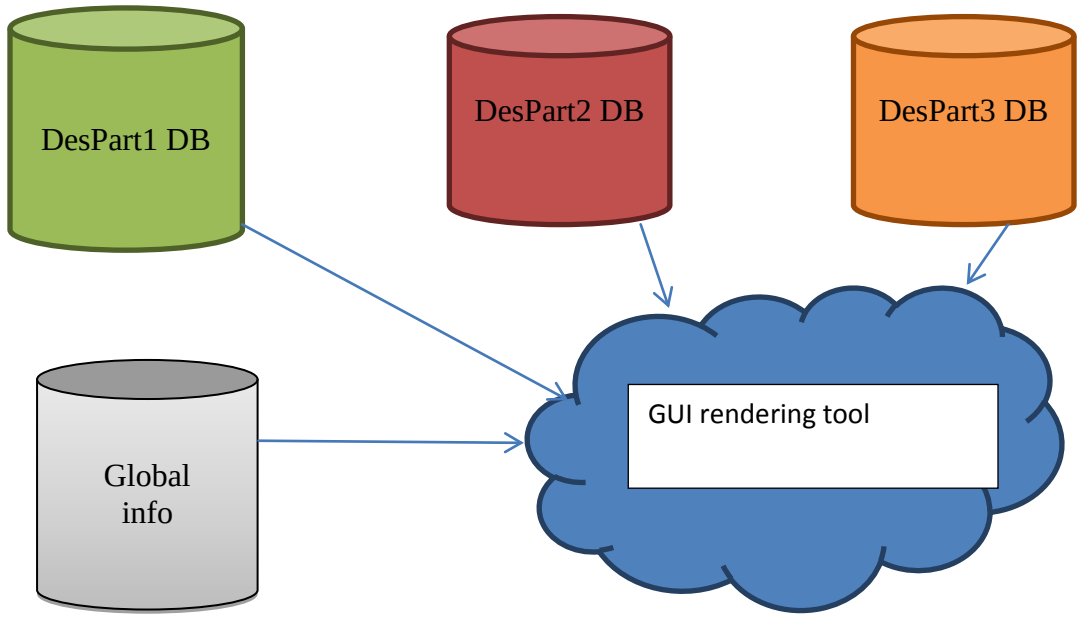


*Figure 2: Debug database structure*

The above picture shows flow of handling debug database by GUI rendering tool. The design is partitioned into Despart1, Despart2 and Despart3. The segregated parts of design produces separate debug databases. Different colors on the databases indicate that the debug data dumped for that part of the design are different. User has flexibility to modulate amount of database that will be dumped for different parts of design. This is the structure of database that the tool needs to handle.

Any standard database tool allows the user to access multiple databases through the handles they supply to user. The user must request the database tool to open a particular database (say db1) for information access; the tool does that and returns a handle (db1_handle). The user then must interact (execute queries to fetch relevant information) with the database (db1) through this handle (db1_handle).

The database structure for debugging as described above will have several databases and thus database handles for each. The top (or global) database will supply the databases with the path to each of these databases; however GUI must handle opening, accessing right database and closing them on its own. For this purpose the associative array of Tcl language comes in very handy once again. One can easily create an associative array for database handles. For example, for the above case discussed, the Tcl storage would be –

```
array set db_handle_array {}
set db_handle_array(db1)  db1_handle
```

Now, one can simply find out which queries should be carried out on which database (say db1) through the global database, and execute the queries on the database handle ($db_handle_array(db1) stored in the associative array. Because the array is associative, the worst time complexity to pick the correct handle is constant.

The second requirement of having a multiple database in a flexible GUI tool is, one opens a database only when that part of the design is accessed. So the database handles are created on the fly while drawing the part of the design that database holds. This again needs to be a fast (preferably in constant time) action for GUI. The GUI must in constant time determine if a database is already opened and handle is available if not then create that handle. Tcl gives a solution through where one can check if there is any value stored in an associative array against a particular key. So the algorithm of handling this flow would be -

```
If { [info exists db_handle_array(db1)] } {
     Use existing db handle
} else {
     Open db handle for db1
     Set db_handle_array(db1) db1_handle
}
```

The last requirement of handling multiple databases is, one must close all these handles before exiting GUI. Any database on which handle is kept open, might not behave correctly if a subsequent process tries to access it. However the flexibility that Tcl offers while accessing associative array fixes this problem. One can easily traverse an associative array like a list. The “array names” functionality brings all the keys of an associative array for the database handle arrays. The flow of closing all databases is –

```
foreach key [array names db_handle_array] {
     Close database whose handle is stored in
$db_handle_array($key)
}
```

**Conclusion:**

Using the above described approaches made as develop an efficient schematic widget tool. IncrTcl helped us in creating the main window providing the needed functionality and storing window specific information locally. Name spaced helped us in providing well defined interface to fetch the required data and to manage a common cache of data. This also helped in keeping the GUI side clean and clear from code to interact with database and keeping the GUI code thin. Associative arrays helped significantly in managing multiple databases parallel and at also helped in caching and retrieving the data quickly and easily.
When all of these constructs of Tcl gets combined, then comes the real power of Tcl through which, however complex the widget is, looks easy and trivial to create and maintain.

**Bibliography:**


[1] An Object Oriented Mega-Widget Set, Mark L. Ulferts, http://incrtcl.sourceforge.net/iwidgets/paper/paper.html

[2] TCL wiki, http://wiki.tcl.tk

[3] Can Distributed DB Provide An Effective Means Of Speeding Web Access Times, Christopher G. Brown, http://jitm.ubalt.edu/XVIII-1/article1.pdf

[4] Using [incr Tcl] to improve stability of a GUI – A Case Study http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2009/proceedings/guis/incrtcl-emulation-debug-gui.pdf