Debug APIs
– next wave of innovation in DV space

Srinivasan Venkataramanan[1] Ajeetha Kumari[2]

[1]Organization: VerifWorks Pvt. Ltd.
2nd Floor, VTD Square, 183/3, Sarjapur Road, Dommasandra, Bangalore, India 562125
Job Title: Verification Technologist, Email: srini@cvcblr.com Mobile no: +91-9620209225

[2]Organization: VerifWorks Pvt Ltd
Job Title: Verification Consultant Email: akumari@cvcblr.com Mobile no: +91-9620209224

Abstract-Debug is one of the least automated tasks in the entire Design Verification (DV) process. With SystemVerilog and UVM providing standard DV language and libraries, it is time for the industry to standardize debug and focus on automating several manual steps involved in debug flow. In this paper authors share their experience of building debug automation using available Debug-API standards. We also share updates on a new industry initiative named DD-API (Debug Data API).

I. INTRODUCTION

Electronics design process has evolved over many decades leading to more and more complex functionalities in a single chip. Such productivity is often achieved through major "shifts" in the level of abstraction, level of automation, structured design frameworks, standard verification methodologies etc. One area that has remained not so well automated yet has been - the debug. Debug may mean different things to different people – just like how verification methodology itself was prior to UVM [1] days.

Since debug is consuming significant portions of a typical ASIC design cycle, it becomes critical to be able to automate many of the mundane tasks. In the past decade or more, some of the low hanging fruits (though critical) such as dumping waves on demand, automating the dump file creation on a simulation failure etc. have been automated. However, there are many other tasks that have remained largely manual to many teams across the world. Part of the problem was there no standard to query the debug database in the past. However, with tools like Verdi offering open API (called NPI [3]) that could work across simulators, this problem is getting addressed by the EDA vendor side. The recent initiative on defining an open "Debug Data API [7]" is another good step in making this process further seamless across EDA vendors. The availability of such API is but only a first-step in debug automation. It is a critical step that opens up the next wave of productivity for designers as several common tasks of debug across teams can be created as "applications" and reused. We will refer to these applications as "apps" henceforth.

In this paper, we share our experience of developing and deploying certain classes of smart Debug "apps" with Verdi (called "VC Apps" [4]). We provide the motivation for few common problems, pseudo algorithm of possible approach to automate a possible solution and results from early stage deployments. We will also share the early sketches of "Debug Data API" as in its current form (Version 0.95 as of the time of this abstract writing) with specific focus on the proposed flow of usage, benefits to end users etc. Given that this Debug Data API is drafted around the hugely successful and popular Verilog's VPI [2] data model; we expect readers to quickly pick-up the core concept. We will share few examples put forward by the Debug Data API team so far to demo end users what could be the next wave of productivity that this Debug Data API can bring to them.

II. Simple Debug Apps – automated, smart debug data generation

Over the years many DV teams have learnt that dumping debug database can be a costly affair. With designs becoming larger and larger, the size of dump file (and associated debug data-structure) keeps growing. Popular EDA tools however provide ways to handle this through several means:

- Split dump files into multiple physical files to circumvent UNIX OS (32-bit) limitations of a single file size
- Dump only relevant portions of the design
- Dump only limited time periods than the entire simulation run
- Re-create some of the combinatorial signals as a post-processing step (Synopsys Siloti [5])

Most of these techniques are automatable via TCL interface of EDA tools.
There are also custom apps to serve specific tasks such as:

- Gate Level to RTL mapping of signals
- Simple X-tracing to follow cone-in of a given net etc.

We would like to refer to these tiny, handy TCL scripts as first generation debug "apps".

II. VC Apps – Regression Test Generator (RTGen)

A. Introduction to Regression Test Generator (RTGen)

A regression test by definition is a stable test that has been developed over some time and has been found valuable by the team to run often during specific milestones. However, given the complex nature of modern day constrained-random tests, the repeatability of these tests may become a concern. Specifically, a test that generated a series of input vectors during one run may not reproduce the same stimulus over a period of time (if the constraints get modified for instance). RTGen [8] fits exactly this gap. Once a stable constrained-random test is found valuable, RTGen helps to extract a directed test from a constrained-random simulation run.

B. Need for extracting testcase from failure dump

In SystemVerilog UVM-based verification, a typical constrained-random test consists of many sequences started in a specific order. Each sequence generates a certain number of sequence items with random values for its members. Once such a test is made stable, the same test is run with several different seeds to leverage on constrained random generator available with EDA tools.

In a typical DV (Design Verification) cycle, as soon as a design error (colloquially known as "bug") is spotted by the DV code, it is very important to retain that trace/series of inputs leading to that failure. Often teams create a dump file as a trace in binary, compact form such as FSDB, WLF, SHM, ASDB etc. The binary dump file is great for analyzing the error, and understand the sequence of events leading to failure. Engineers analyze the dump files and fix the issues inside RTL, provided it is an RTL issue. In this process of debug and analysis, often designers want to re-run the trace. Often RTL engineers responsible for the bug fix are not the ones who wrote the tests. Also, given the complexities around SystemVerilog & UVM, some designers are not comfortable re-running these tests.

The dump files produced by a constrained-random test run is a good source of the stimulus scenario leading to the design errors. However, the dump file cannot be directly re-run as a test to quickly reproduce the same scenario. In a typical SystemVerilog UVM setup this would require saving the seed and constraints and re-run on need basis.

With RTGen, one can extract a directed stream of test vectors from a dump file. Hence a dump file can be quickly converted back to a Verilog/SystemVerilog/UVM sequence without any randomization.

B. Challenges in reproducing scenarios of a failing test

The above requirements are obvious for most involved in the DV process. However, repeating a "test" is not as easy it may seem at the first look. More so in case of popular SystemVerilog & UVM based environments that leverage on constrained random verification technique to automate stimulus generation. While random generation is great, sometimes the repeatability of randomness (also known as “random stability”) becomes a challenge. All simulators support random stability on the same code base given the starting seed. Refer to SystemVerilog LRM [2] for a detailed discussion on random stability and how new object creation and new thread creation can affect random stability. Often DV engineers add extra constraints, add more threads etc. Some of these changes technically modify the state space in such a way that no tool can reproduce the same stimulus even with same seed.

Also, with geographically dispersed teams, some companies prefer to hand-over a directed test to remote debugging teams though the bugs were originally caught using a constrained random test.

Another challenge that DV teams face often is to be able to ensure that for every bug found, there is a test that reproduces the input stimuli leading to the failure. Ideally such bug-finding-tests shall be added to a regression suite and be run at every mile-stone. With DV environments constantly changing during the course of a project, such guaranteed re-runs would typically require directed tests per failure (bug).

One approach to solve this problem is to create a “directed test” that reproduces the same trace as the random test (with a given seed) that found the bug. Availability of such an extracted “test” is critical for variety of reasons including:

- Designers will need to re-run the trace few times to analyze the design, fix etc.
- DV engineers can confidently add this test to their regression suite and ensure that the same error does not creep-in later in the design cycle.
- Management wants to ensure all such past bugs have been re-verified and do not re-enter

The downside of this approach however is that one has to recreate the input scenario leading to the bug. This can take significant time by the DV engineer.

Another approach that is followed is to retain the randomness of the test, but add a functional coverage point to capture the scenario and then let the randomness hit those points. This will ensure that any bug related symptoms are re-run at every major mile-stone in the future by indicating a coverage hole if failed to do so. However, this option means lot more extra work for the DV engineer.

Figure-1 below captures the traditional flow of ensuring a design error (“bug”) gets reproduced in regressions.

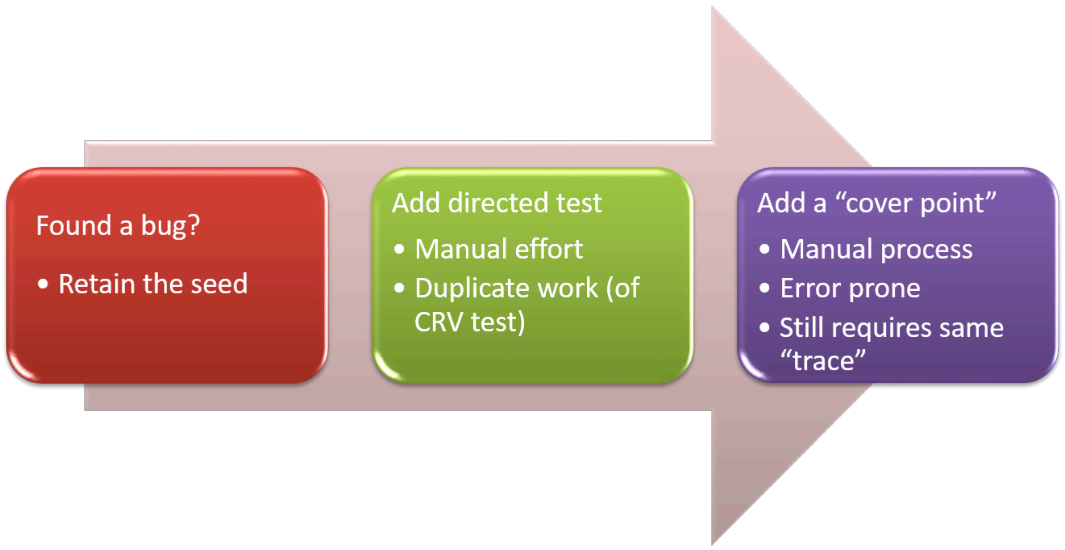


Figure 1. Conventional Regression flow to ensure bug reproduction

The manual conversion of a random test to a directed test and/or a functional coverage point is a time-consuming task. It is also error prone as the scenarios can be fairly complex with 1000s of clock cycles. Often the management sees this as a duplicate effort as the bug has already been found and being fixed by RTL team.

Given that most of the bug reports include a corresponding debug database (dump files), a smart "app" can be created to extract time-value pair of all primary inputs. Once the time-value pair of all primary inputs are extracted to a database, it can be translated back to a directed "trace" in the form of Verilog, SystemVerilog etc.

This "app" is intended to extract a directed test from a constrained random test setup. Given a waveform file from a constrained random test, this "app" extracts all relevant signals. Then it extracts time-value pair of each signal change. Once the data has been mined, it can be exported to a directed test/sequence in various flavours including Verilog, SystemVerilog, UVM, graphs etc. This app has been successfully used in projects using as constrained random simulation.

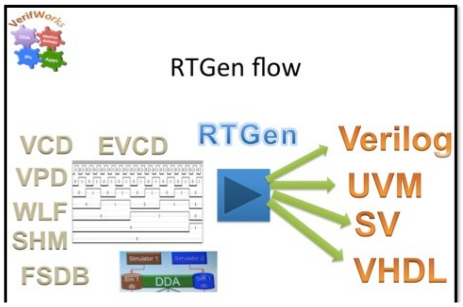


Figure 2. RTGen flow

Synopsys' Verdi provides a well-structured and well documented API (with object & data model similar to that of popular Verilog VPI). It is convenient to develop an application ("app") to walk through signals of interest and extract time-value pair information for various kinds of data-types.

The output from this app can be generated in various formats including:

- Verilog (plain V2K/Verilog-2005)
- SystemVerilog (without UVM)
- SV + UVM (with open-source Go2UVM test layer around, see [4])
- SV + UVM as regular UVM sequences (and a testbench layer around to run multiple sequences generated from multiple FSDB files for the same design)
- VHDL

A. Architecture of RTGen VC app

RTGen is implemented as an “app” on top of Synopsys Verdi platform. With Verdi, an API is available to query the data structure and time-value pairs of signals inside the database. This API is called as NPI (Novas Programming Interface [3]).

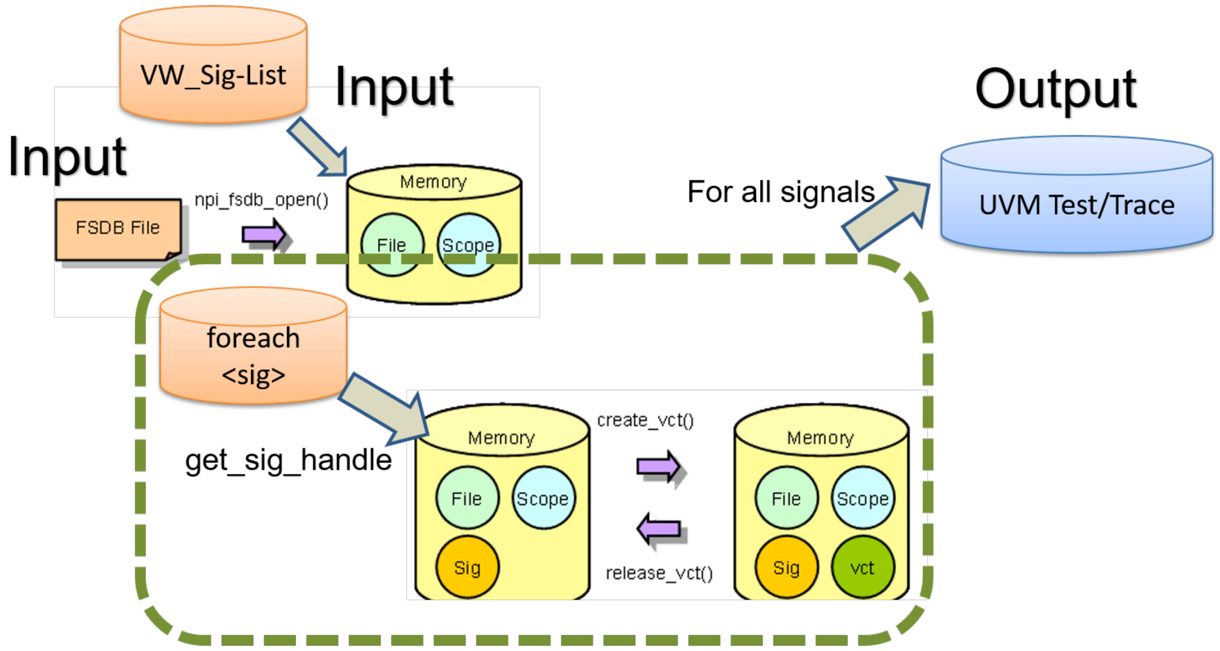


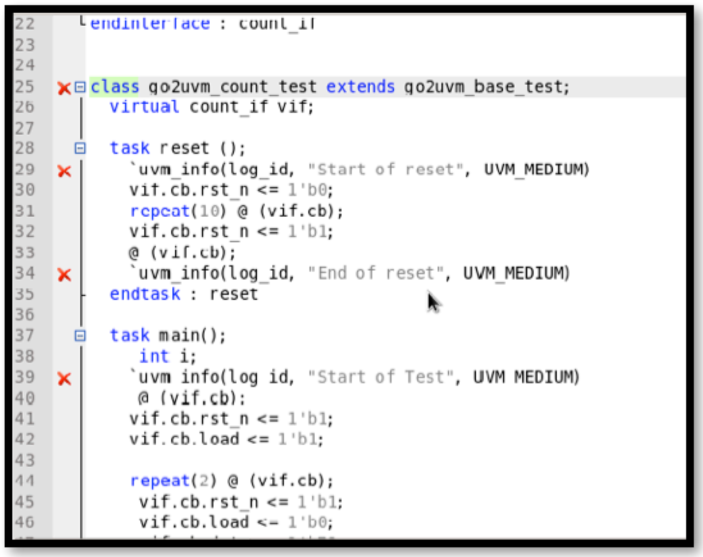
Figure 3. Architecture of RTGen app with Verdi

Inputs to the app are the FSDB file and the signal list (DUT input signals alone). The app reads the FSDB into the program memory, then reads the signal list one-by-one. For each signal the app gets the signal handle through Verdi’s NPI. Once a handle is found, it is stored in a list for future reference. The signal handle is used to create a Value Change Traverse (VCT) object and each transition is captured in a value-time pair. This data is stored in a database/hash-table.

Once all signals are parsed, and the VCT table is updated, the app creates a SystemVerilog interface file containing all signal names. It then creates a skeleton UVM test using the open-source Go2UVM package to keep the UVM code minimal.

A variant of this app also creates a full-fledged UVM framework with driver, sequencer, agent, env, sequence and test. In this mode, the app can create one sequence per FSDB for the same design – hence easing the regression runs for the DUT later in the design process.

For teams using UVM, it is often desired to extract a test in familiar UVM format itself. Though UVM sounds like an overkill to re-run failing traces as it requires good OOP understanding, an open-source test layer has been made available named Go2UVM [6] that eases this pain. With Go2UVM users can write a test as simple as they are used to in plain Verilog and the typical complexities of constructing the component hierarchy, phasing methods, raising and dropping mandatory objections etc. shall be taken care of by the open-source Go2UVM package. A typical RTGen extracted test using Go2UVM looks like below:



```
endinterface : count_if

class go2uvm_count_test extends go2uvm_base_test;
  virtual count_if vif;

  task reset ();
    `uvm_info(log_id, "Start of reset", UVM_MEDIUM)
    vif.cb.rst_n <= 1'b0;
    repeat(10) @ (vif.cb);
    vif.cb.rst_n <= 1'b1;
    @ (vif.cb);
    `uvm_info(log_id, "End of reset", UVM_MEDIUM)
  endtask : reset

  task main();
     int i;
    `uvm_info(log_id, "Start of Test", UVM_MEDIUM)
     @ (vif.cb);
    vif.cb.rst_n <= 1'b1;
    vif.cb.load <= 1'b1;

    repeat(2) @ (vif.cb);
     vif.cb.rst_n <= 1'b1;
     vif.cb.load <= 1'b0;
```

Figure 4. A simple Go2UVM test

III Applications of RTGen VC app

A. RTGen in a Constrained Random UVM flow

In a typical Constrained Random Verification setup with SystemVerilog (with or without UVM), common flow to handle a bug is as shown in Figure-1 (above). With RTGen VC app, the manual effort is reduced and the flow looks as below:

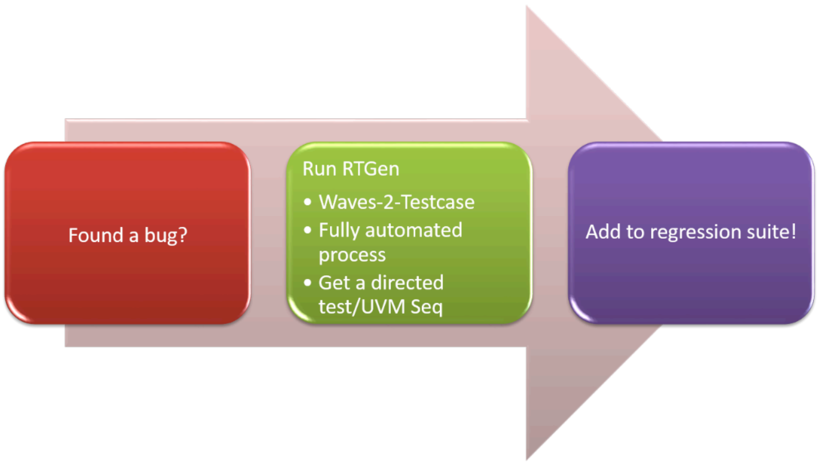


Figure-5. RTGen in constrained random verification flow

A sample waveform is shown below in Figure-6.

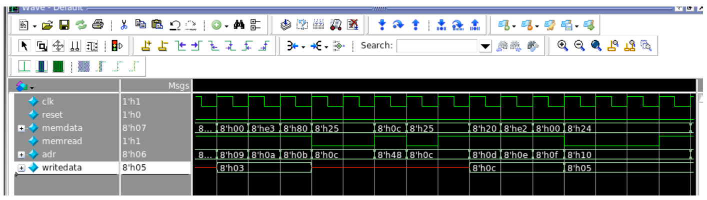
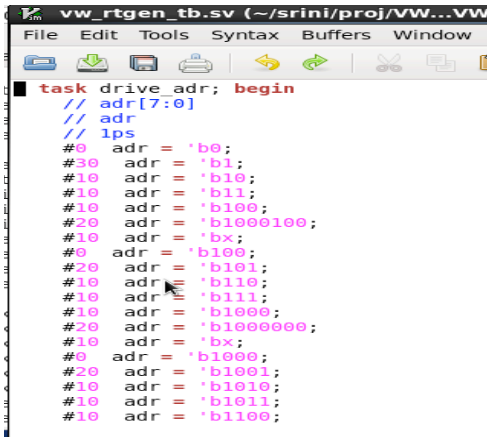
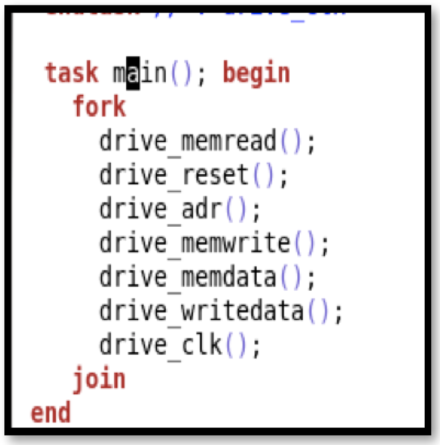


Figure-6. Sample waveform as input to RTGen

A plain Verilog test extracted from the waveform above is shown in Figure-7 below:


vw_rtgen_tb.sv (~/srini/proj/VW...VW
File Edit Tools Syntax Buffers Window

```
task drive_adr; begin
   // adr[7:0]
   // adr
   // 1ps
   #0  adr = 'b0;
   #30  adr = 'b1;
   #10  adr = 'b10;
   #10  adr = 'b11;
   #10  adr = 'b100;
   #20  adr = 'b1000100;
   #10  adr = 'bx;
   #0  adr = 'b100;
   #20  adr = 'b101;
   #10  adr = 'b110;
   #10  adr = 'b111;
   #10  adr = 'b1000;
   #20  adr = 'b1000000;
   #10  adr = 'bx;
   #0  adr = 'b1000;
   #20  adr = 'b1001;
   #10  adr = 'b1010;
   #10  adr = 'b1011;
   #10  adr = 'b1100;
```



```
task main(); begin
  fork
    drive_memread();
    drive_reset();
    drive_adr();
    drive_memwrite();
    drive_memdata();
    drive_writedata();
    drive_clk();
  join
end
```

Figure-7. RTGen extracted test

The above test shown is in Verilog format. We can also generate in various other formats. In a nutshell, RTGen is useful to replay constrained random tests by extracting a directed trace/UVM sequence.

B. Replaying formal verification failures in simulation/regressions

A typical formal verification (model checking) involves no testbench, just the RTL and the assertion (and assumptions) model. When a FV tool finds a failure, it generates a dump file such as FSDB. Designers can then use a waveform viewer such as Verdi to view the FSDB to understand the failure and fix the RTL. However, given that the FSDB is not a replay-able test, it is hard to add such failing traces to regressions. It is often desired to bring FV failures to simulation, some of the common rationales include:

- Simulation tools tend to provide better debug capability especially in interactive debug flow – such as breakpoints, expression builder etc.
- At times the FV failures may need to be analyzed by simulation teams to understand any potential gaps in their flow and improve.
- FV failure traces can be a great source of regression tests for simulation flow.
- Developing CIP (Checker IPs – subset of a typical VIP with assertions/checkers alone) for formal and simulation compatibility requires cross testing of the CIP model and validate the failures in both Simulation & Formal.

As RTGen converts any FSDB to an equivalent Verilog/SystemVerilog/UVM trace it fits nicely in this flow.

IV Limitations of RTGen VC app

While RTGen can be handy in stable regression setups there are certain restrictions. The primary one is the need for a stable design atleast on the primary input-outputs (IOs). Since RTGen extracts vectors from a dump file and applies the vectors back to the design, it is essential to have stable signal names, widths etc. But practically speaking this is not a big issue as RTGen is supposed to be used in a project after a stable RTL is available.

Another important limitation is the timing of primary IOs. In case of certain error fixes, the RTL might change the timing of its outputs. If the input signals are dependent on the previous value of output signals, it will impact the RTGen timing and render the generated tests unusable.

Another limitation is on the use of PLI/VPI based drives to internal nodes of the design. Currently RTGen does not take care of any non-IO signal changes, though a support is planned for future versions.

Another limitation (minor) is that though the generated test is readable and well structured, it is hard to edit is manually. This is because the generated test is a simple linear test with one thread per signal. Given the incremental, linear delay between each value change, any manual changes to the threads is likely to change the test intent. We consider this as minor limitation as the generated tests are expected not to be tweaked, instead the recommended approach is to regenerate a new test from a modified dump file.

V Quick introduction to data model of Debug Data API (DD-API)

Debug Data Interface (DDI [7]) is a procedural interface that allows programs implemented in standard programming languages, such as C or C++, to access waveform data generated from simulation or emulation of electronic circuits as well as design data. The popular Verilog Procedural Interface (VPI) has a wealth of design and verification data-access mechanisms. The DDI is an extension layered on top of a subset of the existing VPI. The intent is to use the VPI to traverse the design and retrieve data in the same manner that one would use the existing VPI currently with an active simulation. However, VPI access to simulation control, and writing data would not make sense in a post-processing environment. DDI is an ideal vehicle for tool integration in order to replace arcane, inefficient, and error-prone text file-based data exchanges with a new mechanism for tool-to-tool and user-to-tool interface. Moreover, a single-access API eases the interoperability investments for vendors and users alike. Reducing interoperability barriers allows vendors to focus on tool implementation. Users, on the other hand, are able to create integrated design flows from a multitude of best-in-class offerings spanning the realms of design and verification; such as simulators, debuggers, and formal analysis, coverage, or testbench tools.

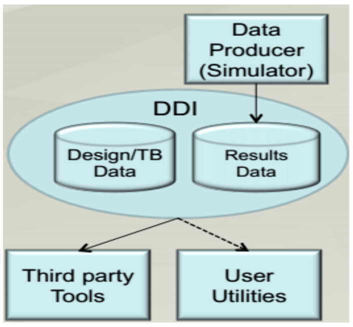


Figure 6. DD-API conceptual diagram

VI Summary and future work

Ability to develop "apps" to automate certain mundane tasks in design verification is encouraging. With emerging standards for debug, such automation will further help tackle the ever-increasing complexities of design verification process. In this paper, we have shared our experience with first generation debug apps. We then provide details on motivation behind developing RTGen as a debug app on top of Synopsys Verdi. While NPI is still a single vendor implementation, the upcoming DD-API is worth watching. We will be keen on porting our RTGen to DD-API as soon a stable version becomes available.

ACKNOWLEDGMENT

REFERENCES

[1] Accellera UVM - http://accellera.org/downloads/standards/uvm
[2] SystemVerilog LRM - http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
[3] Novas Programming Interface (NPI) - https://www.synopsys.com/Tools/Verification/debug
[4] Verdi's VC Apps - https://www.vc-apps.org/SitePages/Home.aspx
[5] Siloti - Visibility Automation System https://www.synopsys.com/Tools/Verification/debug/Pages/siloti-ds.aspx
[6] Go2UVM open-source test layer, www.go2uvm.org
[7] Debug Data API Google group - https://groups.google.com/a/debugdataapi.org/forum/?hl=en#!forum/dd-api
[8] RTGen - http://verifworks.com/products/rtgen/