

# Vesper: Verilog Signal Profiler

Ilya Wagner

Advanced Computer Architecture Lab  
The University of Michigan – Ann Arbor, MI  
iwagner@umich.edu

## ABSTRACT

Verification of complex hardware designs remains a very labor-intensive process often involving multiple designer and verification teams. Unfortunately, often the designs under verification are poorly documented and invariants of circuit operations are ill-defined increasing the burden on testers and verification engineers, who must understand internal aspects of the design written by other people. Since forcing the design team to write complete and exact specifications for each submodule is extremely costly and counter-productive, it is much more effective to help engineers to deduce internal design invariants from interface level behavior. This paper presents Vesper (VERilog Signal ProfILER) - a tool that identifies internal design signals causing interface level behavior through signal value profiling. Vesper engine automatically extracts internal design signal from register-transition level Verilog and creates monitoring environment based on user-described interface events. After running several testbenches the analytical engine of Vesper reasons on which signals could have caused the events to happen based on the frequency of the observed signal values. Vesper is very flexible and can be parameterized through several pre- and post-simulation filters to effectively limit the number of signals reported and thus help verification engineers in understanding the internal structure of the design. Our tests show that Vesper is impervious to false-negatives and is quite precise even with a small number of testbenches. We also postulate how Vesper can be used in verification as a part of a directed random simulation engine, making much of this work automatic and increasing productivity of the verification process.

## 1. INTRODUCTION

Verification remain the most labor consuming phase of the hardware design process. It is estimated that about 70% of overall design and development effort is spent on verifying the functionality of the hardware. It is a common practice to have separate teams performing design and verification of the sub-components of the design [3]. Although this approach has its advantages, it also has a fair share of challenges. The most prominent of them is, no doubt, familiar to all readers with even minimal programming background - it is hard to debug other people's code. Unfortunately this is often a big part of a verification engineer's job, especially in initial stages of the testing process. In order to create bug-free testing and verification environment and filter out false-negative bugs, arising from incorrectly designed

testbenches and simulation settings, a verification engineer must be closely familiar with the internal structure of the hardware. Only then it is possible for him or her to see if erroneous behavior of the design is genuine, without requiring extensive assistance of the designers, who wrote the code. This comprehension of the internal structure of the design is very dependent on the naming convention, which, unfortunately, might vary between individual submodules making the work of a verification engineer an order of magnitude harder. Besides making each designer verify his or her work and actively participate in the system level testing, there are only a handful viable solutions to this problem. One approach is to require all designers to follow strict naming convention, extensively comment their work, and provide good explanations of decisions made. This, however, is hard and probably not economically feasible to enforce, since it requires stringent human control. Another solution is to try to automatically identify only the relevant part of the structure and internal signals for a given behavior of design. For example, when a verification engineer sees an error of the data bus interface of the design during a read transaction, he or she can only investigate the logic driving the read control signals instead of the whole interface FSM. This solution seems more viable due to its automatic nature and independence from human factors.

In this paper we present a signal profiling tool for Verilog called Vesper (VERilog Signal ProfILER). By collecting and analyzing the logs of testbench runs Vesper tries to identify signals within a design whose behavior has high correlation with the user-specified events. Vesper sets up most of the framework required to run the testbenches, produces the log files, and does the analysis automatically.

Vesper was first conceived as a plug-in for StressTest[2] for automatically determining activity signals for monitoring. The activity monitors in the StressTest analyzed the behavior of these key signals for directing the test generator towards interesting (and prone to bugs) areas of the design functionality. However, initially these signals were hand-picked, which required knowledge of the architecture of the design under verification (DUV). And, although StressTest showed very promising results in bug coverage, its performance was heavily dependent on the signals picked, and thus the comprehension of the design by the verification engineer. From this observation the idea of using log analysis and signal profiling for identifying the candidates for activity monitoring was born, and work on Vesper began. However, the experimental results showed greater applicability of this approach and flexibility of Vesper software in other verification applications, discussed below.

## 1.1 Contributions

The main contribution of this paper is the development of highly portable and flexible signal profiling methodology for designs written in Verilog HDL. In addition, the paper presents an innovative text-based approach to describing synchronous interface-level events of interest. The events are described based on values of signals (called triggers) before, during, and after the events' occurrences. This approach of identification of key control signals is almost fully automatic and doesn't require detailed knowledge of the DUV architecture. Moreover, the described methodology allows porting the results of profiling into a variety of tools, which further simplifies the whole verification process.

The remainder of this paper is organized as follows. Section 2 describes the structure of Vesper and the approach it uses for signal profiling. Section 3 performs a study of the performance of the tool, by conducting experiments on two sample designs. Finally, Section 4 draws conclusions, identifies applications crucial to verification that might benefit from such a tool and suggests directions for future work.

## 2. VESPER TOOL

Vesper provides an efficient and simple to set up framework for analyzing designs written in Verilog HDL and helps to find signals highly correlated with interface events. Vesper event specification language provides a flexible way to describe the events of interest through sequences of signal values at specific times. An advantage of Vesper is that given the Verilog code for the design and text-based event description it automatically sets up most of the testing framework, runs selected testbenches, and analyzes the results, demanding minimal interaction with the user. This section gives an overview of the tool's structure.

### 2.1 Overall Structure

Vesper is composed from three main modules: Verilog parser, Event parser, and Analyzer, illustrated on Figure 1. The former two modules perform preprocessing and setup of the testing framework, while the later one analyzes the the results of the runs and actually does the signal profiling. The user input to Vesper consists of Verilog design files, filtering lists and event description file. Using these inputs Vesper constructs a custom Vera shell that connects to the test environment, identifies the events of interest and records the values of signals (the system snapshot) during the simulation. We chose Vera [1] rather than System C or other language for this sampling since it is much easier to bind to arbitrary locations anywhere in the DUV and it allows for writing multi-thread programs that can monitor multiple events simultaneously. The collection of all snapshots taken at occurrences of a particular event is then passed into analyzer for value frequency profiling. The end user can affect the precision of the analyzer by changing the selection threshold and/or invoking optional value filters. The output of the analysis is a text based description of signals that Vesper deems to be correlated with given events. Optionally Vesper can report the results in a form of Vera interface to the design. This option is provided for usage of Vesper with StressTest, since the later tool requires an interface description of the activity monitors for directing the test generation. Vesper is written in C++ and contains about 2000 lines of commented code.

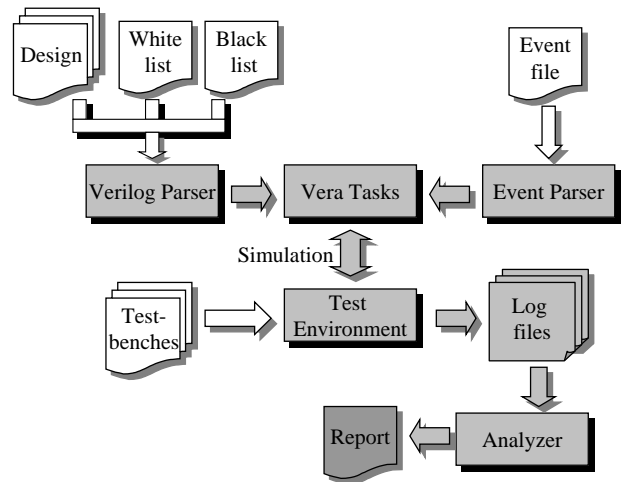


Figure 1: Vesper structure

### 2.2 Verilog Parser

Verilog parser is designed for extracting signal names from a set of Verilog files. A signal in the described framework is an input or an output of a module, or a wire or a register (reg) inside the module. As shown in Figure 1 the input to the parser is a set of Verilog files and filtering lists and the output is a custom Vera interface. The parser does its work in several phases.

**Signal and instance extraction** is the first phase of the parser's work. The tool goes through each Verilog file individually and considers separate modules described in these files. Note, that it is a common practice among designers to declare multiple modules in one Verilog file, especially when the modules are relatively small and related by functionality. The parser records all declared inputs and outputs of the module, all internal wires and register. In addition to that the parser records names of other modules instantiated from the current one. This later be used in the signal naming process, that traverses the whole hierarchy of the design.

It was our design decision to make the parser ignore arrays of registers used, for example, in register files, FIFOs, and so on. Virtually in all of the designs these registers reside on the data-path and have little to do with the control of the design by themselves. And even if the register arrays are involved in the control-path they will have a set of wires or outputs (read ports) emanating from them that will be picked up by the parser.

**Duplicate name elimination** is conducted on the set of recorded signals, individually for each module. Duplication of names is often used in Verilog to specify implicit connections between different types of signals. For example, if a design has a register *empty* and an output *empty* in its interface, there exists an implicit connection between those two signals. It is easy to see that instances with duplicate names within one module always have the same value and thus are redundant and all but one of them can be eliminated.

**Signal naming** is done to create exact hierarchical names for each signal. The user specifies the top module of the design where naming process starts. After naming the signals in the topmost module the parser proceeds down the hierarchy tree. Note that submodules in Verilog designs can be instantiated multiple times at different hierarchical levels and serve for different purposes. So, internal signals inside of these instances might be controlling different actions and need to be identified and uniquely named.

**List application** is an optional phase that is used to reduce the size of the system snapshot that will be taken during the simulation and improve precision of the analysis. As it is illustrated in Figure 1 the user can provide the Verilog parser with the Black and White lists of signal names. The Black list includes strings that user knows are present in *uninteresting* signal names. If a signal name contains one of these black-listed strings the signal will not be considered in later analysis. This allows the user to remove, for example, clock and reset signals and even known portions of the data-path from the list of signals being analyzed. On the other hand, the signals in the White list must be present in the final report. These signals are eliminated from the list due for analysis, but are recorded on the side and later added to the final report by the analyzer.

**Vera interface generation** is done after all relevant signals are uniquely identified. From a list of signal names the parser generates a custom Vera interface that binds to those signals in the DUV. Also the parser creates a storage class that has methods to save and record to file values of all the signals, *i.e.* take the snapshot of the system.

<pre> <b>module</b> rf (CLK, RSaddr, RDaddr, RT, RD); <b>input</b>    CLK; <b>input</b> [4:0] RSaddr; <b>input</b> [4:0] RDaddr; <b>output</b> [31:0] RT; <b>input</b> [31:0] RD; <b>wire</b> RWrite; <b>assign</b> RWrite = (RDaddr != 5'b0);  <b>reg</b> [31:0] RAM [0:31]; <b>reg</b> [31:0] RD, RT; ... <b>endmodule</b> </pre>	<table border="1"> <tr> <td><b>Black List</b></td> </tr> <tr> <td>CLK</td> </tr> <tr> <td><b>White List</b></td> </tr> <tr> <td>rf.RDaddr</td> </tr> </table>	<b>Black List</b>	CLK	<b>White List</b>	rf.RDaddr
<b>Black List</b>					
CLK					
<b>White List</b>					
rf.RDaddr					

---

```

interface activity
{
input [4:0] dut_RSaddr PSAMPLE #-1 verilog_node "dut.RSaddr";
input    dut_RWrite PSAMPLE #-1 verilog_node "dut.RWrite";
input [31:0] dut_RT PSAMPLE #-1 verilog_node "dut.RT";
input [31:0] dut_RD PSAMPLE #-1 verilog_node "dut.RD";
}

```

**Figure 2: Example of generated Vera interface**

Figure 2 provides a short example of using the parser on a sample register file design. For simplicity this example doesn't include any hierarchical traversal. The Black list for the design contains a string *CLK*, which names the clock signal, while the White list contains description of signal *RDaddr* in register file module. The result of the parsing is

the Vera interface is also shown in the Figure. For purposes of saving space the storage class description and methods are not provided here. Note that register array RAM is not included in the final output, as it is merely a portion of the data-path. Also notice that signals *CLK* and *RDaddr* are not included in the output since the first one of them was black-listed and the second one appeared in the White list and was recorded in a separate file. Finally, observe that names *RT* and *RD*, which are instantiated as both registers and ports of the design, appear only once in the interface.

This interface, along with the storage class, provide only a half of the needed Vera functionality. The rest of the functions, which are responsible for identifying the events of interest and recording the system snapshots through calling the storage class *save* and *print* methods, are generated by the event parser. The following section will describe the format of the text-based event description and then the work of the event parser will be illustrated.

### 2.3 Event Description and Parsing

The second pre-processing module of Vesper is the Event parser. The only input of this parser is the text description of events that need to be correlated with the design's internal signals. For example, when investigating memory bus interface of a processor the user might want to find out which signals influence the write command. Then the user is basically interested in an event *WRITE* which occurs when a write command is put on the memory command bus on a rising edge of the clock.

To describe this and much more sophisticated events Vesper provides a simple, yet powerful language that represents the events by a sequence of triggers, *i.e.* specific values of interface signals at particular clock cycles. Figure 3a) presents examples of a simple event *REGISTER\_3.WRITE*. The description consists of a unique event name, the relative clock count at which this events occurs, and a list of triggers, consisting in this case of only one trigger, which is the specific value of the *RWrite* signal. Each trigger also must have a relative clock count value associated with it. In the example in 3a) the relative clock count value for the first and only trigger is 0. Thus, event *REGISTER\_3.WRITE* is said to occur whenever signal *RWrite* has a value of 3 on any rising edge of the clock. Note that the clock count in this case is relative, in other words it is only used to order the sequence of triggers and the event occurrence of this particular event. Figure 3b) shows an example of the simulation when event *REGISTER\_3.WRITE* occurs.

Example in 3c) is a more complex event *SINGLE\_READ*. This event is said to occur when the following sequence of values appears on the *Dread* output of the design *cpu*: 0 on the first clock edge, 1 on the second rising clock edge, and 0 again on the third rising clock edge. However, note that this event actually occurs on the second clock edge. In other words the user wants to know the state of the system on the second clock edge provided that the entire pattern was observed. Figure 3d) presents an example of the occurrence of this event. Note that in this case event *SINGLE\_READ* occurs only at the first time when *Dread* is high. When read is high for the second time it lasts longer than 1 cycle, and the value of the *cpu.Dread* signal is not 0 on the third edge. Therefore, the event is not triggered at this point. Neither it is triggered on the next cycle, since again the trigger pattern did not match the one in the event description.

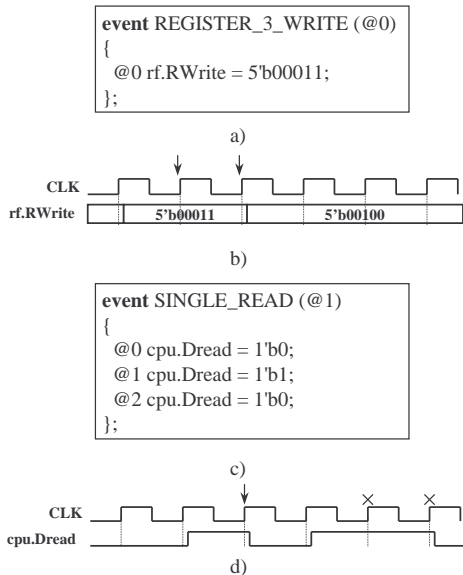


Figure 3: Example of event descriptions (a,c) and event occurrences (b,d)

As the Verilog parser the module for parsing event descriptions is written in C++ and uses similar basic text parsing framework. After all the events are parsed and checked for correctness (presence of the zero-trigger, unique name and so on) Event parser creates a set of Vera tasks that are responsible for monitoring for the events and recording the system snapshots when they occur. The main dispatching task runs in an infinite loop until the simulation terminates and is responsible for monitoring the zero-triggers of events and spawning new Vera threads when they are observed. The spawned threads will watch for all of the triggers of a particular event in order, except the zero-trigger. Whenever a trigger is not matched the thread terminates. In addition these tasks create storage objects and save the system snapshots when the event is said to occur. However, only if the whole sequence of triggers completes the stored state is written to the event's log file. Figure 4 shows a sample event of interest as well as the dispatch and event monitoring tasks written for this event.

## 2.4 Simulation Environment

Simulation environment of Vesper is a simple Verilog testbench that instantiates the DUV and Vera shell and runs one testbench. Note, that unlike in StressTest and some other tools, the Vera module in Vesper is used only for monitoring of the design's state, and cannot determine when a testbench must start or when the run completes. Thus the test environment itself must have means of starting and terminating the simulations in the order specified by the user. These are, however, the only two responsibilities of the test environment, since the rest of the functionality, namely monitoring for the events and recording system snapshots, is provided automatically by Vesper.

Testbenches that are run to generate events of interest can be behavioral Verilog modules or machine code programs, when processor cores are verified. One general requirements to tests is that they should try to exercise various aspects of behavior of design. Usually such tests are available as a part of the testplan or they can be a set of benchmarks for

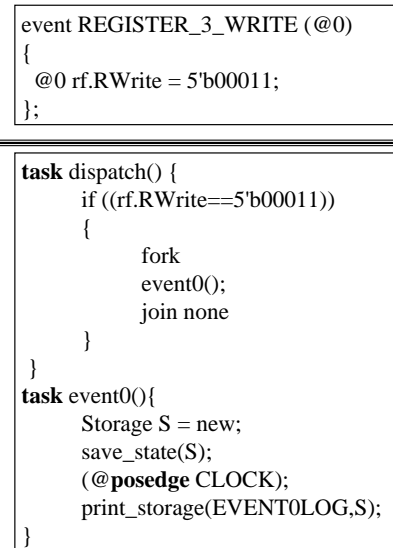


Figure 4: Example of event parsing

a particular design. It is easy to show, that tests exercising different behaviors and modes of operation the design, are more likely to expose different values of irrelevant to the event signals. As a consequence, the analyzer is more likely to select only the signals relevant to the event.

## 2.5 Analyzer

Analyzer engine is the portion of the Vesper tool that actually does the profiling of signal values. It is invoked after all the testbenches complete and the system snapshots are written in the log files, one file per event. Considering each of these files separately, the analyzer calculates frequencies of particular values of each signal in the log. Note that these values are recorded only when a particular event occurs, so, values that are frequent in the log are likely to cause the event. Analyzer filters out instances where signals had undefined  $z$  or  $x$  values, and identifies all the values with frequencies higher than a user specified margin. Note that when this margin is higher than 0.5 at most one value is reported. Signals with at least one highly correlated value are then reported to the user in the form of either a text file that identifies signals, values, and value frequencies, or Vera interface, that identifies only signals. In the later case the white-listed signals set aside by Verilog Parser are added to the interface automatically. Another parameter of Vesper is the *number of valid samples* needed to make a conclusion about the signal-event correlation. The higher is this parameter the more precise results are; however, with higher number of samples required the likelihood of not having enough samples to deduce the correlation grows.

One of the advantages of Vesper is that if the log files are preserved, the analysis can be conducted with different margin parameter without the need to rerun the simulations. In other words, the user can refine the analysis and make the selection criteria tougher and observe signals with higher correlation with the events. Or the user might want to lower the *number of valid samples* parameter if he or she observes that no correlation was established for some of the events of interest.

### 3. EXPERIMENTAL RESULTS

In this section, we first introduce our experimental evaluation framework, and the designs we tested in detail. Then, we evaluate the correctness of Vesper through several iterations of the experiments. In addition to that we analyze the precision of the tool as a function of the number of snapshots taken. Finally, we present the results of profiling of the same snapshot pool with different selection threshold values.

#### 3.1 Experimental Framework

To test the performance of the system we conducted a series of simulation tests on two Verilog processor core designs. The first one is a 5-stage DLX pipeline running MIPS-Lite ISA with branches resolved in ID stage. The second design is a 5-stage pipeline running Alpha ISA with branches resolved in EX stage and two-cycle stores.

For each of the cores we created set of seven benchmarks that are listed in Tables 1. The benchmarks are short assembly programs that test various aspects of core operation, like memory access, branching, computation, etc. The environment in both cases consisted of a Verilog testbench that had a processor core, behavioral data and instruction memory arrays interfaced with the core, and Vera shell monitoring the core. Externally specified program binary file was loaded in the memory at the beginning of simulation and the testbench was executed until halting condition became true.

#### 3.2 Results

The first and the most important experiment was evaluating the correctness of the results produced by Vesper. We ran the tool with all seven testbenches for each of the cores and threshold value set to 1.0. The Black list for both of the cores was minimal and included names of the clock and reset signals. The White lists were left empty. With these settings the Verilog parser isolated 240 unique signal names for Verisimple and 124 signals for DLX. The two events that were monitored for each of the cores were data-memory reads and writes not followed by any other data-memory operation. In other words, if two writes were to occur back-to-back, Vesper only would pick up the later one. Theoretically with these settings Vesper would pick up only signals that caused the memory interface events, however, for Verisimple Vesper reported 72 for write event and 116 signals for read event. For DLX the numbers were respectively, 21 and 24. This was clearly much higher than the true number of signals that were highly correlated with the event.

To lower the number of signals reported for the Verisimple core we decided to extend the Black list to include several signal names that were known to be only a part of the datapath of the core. This reduced the initial signal pool to 198 signals and final report contained 56 and 103 signals for write and read events, respectively, which was also too high.

To understand the nature of this result we investigated the values of the signals reported by Vesper. We found that in the second experiment for Verisimple 45 signals had a consistent value of 0 at the time when write event was occurring and only 9 signals had values other than zero. The signals with value equal to zero were clearly not related to the event occurrence, but were rather indicating correct operation of the pipeline (no stalls and squashes). Similar situation was observed for the read event where 89 signals had value of

DLX testbenches	
AddArr	Adds elements of two arrays
Bubblesort	Iterative bubble sort of an array
FibN	Computes N first Fibonacci numbers
FibRec	Recursively computes Fibonacci numbers
Mult	Multiplies two integers
DotProd	Dot product of two vectors
CalS	Simple stack based calculator
Verisimple testbenches	
btest1	Branch prediction hammer
cache1	Cache hammer
copy	Copies an array
evens	Computes N first even numbers
fib	Computes N first Fibonacci numbers
mult	ROB hammer
parallel	Compute N multiples of 8

Table 1: Testbenches for DLX and Verisimple cores

zero in the final report. When on the third run the analyzer was instructed to remove all signals with zero values from report only the 9 non-zero signals were reported for the write event and 14 for read event for Verisimple and 14 and 16 signals for DLX’s writes and reads.

The following manual check of both cores revealed that all signals that actually drive the event, namely memory interface control wires, were indeed included in Vesper reports, *i.e.* there were no false-negatives. However, the tool reported several false-positives, because some signals in the design either always have a constant value or change the value in rare corner cases not covered by the testbenches. We believe that with a sufficient number of tests and additional work on identifying signals that don’t change their value the precision of Vesper could be increased even more.

The next experiment was designed to analyze the effectiveness of Vesper with varying number of testbenches. The threshold was set to 1.0 and DLX core was used in these tests. The number of signals reported by Vesper after each subsequent testbench run and the number of samples, the number of occurred events was recorded again for write and read events. The results of the experiment are shown in Figure 5. Note that during the run of the first testbench only one event of each kind was observed and Vesper did not report any signals due to insufficient number of samples. The results of this experiments show that Vesper indeed improves the accuracy with more experimentations run.

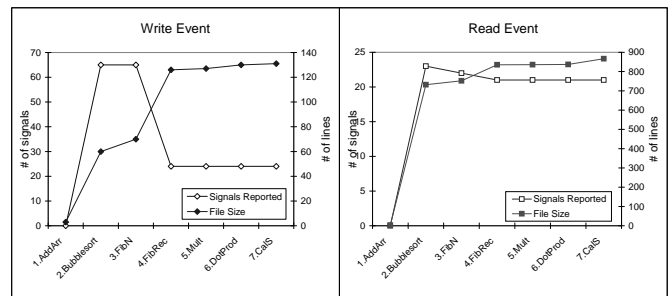
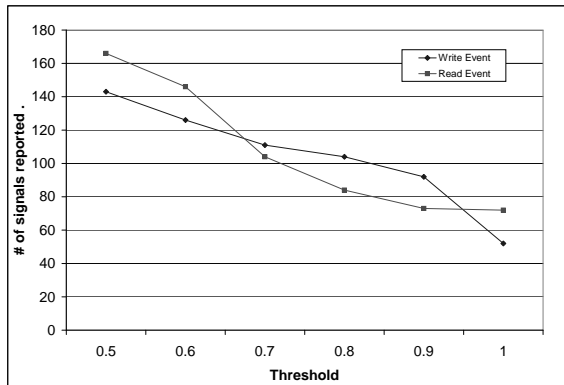


Figure 5: Number of samples and signals reported vs. number of testbenches used

The third and final experiment consisted of the analysis of logs produced by the Verisimple core with different selection threshold values. As it is shown in Figure 6 the higher thresholds correspond to stricter selection criteria, so a much smaller number of signals would be reported.



**Figure 6: Number of signals reported vs. threshold value**

The results of these experiments confirm that Vesper correctly identifies all signals related to the event. Moreover, with increasing number of testbenches and several optional filters the accuracy of the tool grows significantly. Note also, that Vesper never excludes signals with consistent values from the report. Therefore, given accurate definition of events, it will never mistakenly omit a signal directly causing the event from report after any number of tests run.

#### 4. CONCLUSIONS AND FUTURE WORK

Profiling of signal values is a very useful technique for identifying internal properties of a hardware design from interface level events. It aids both designers and verification engineers, and provides a less expensive alternative to requiring programmers to explicitly state design invariants, properties, and keep extensive documentation of work done. Verilog Signal Profiler tool that was presented in this paper can automatically extract signals from complex Verilog designs and set up the test environments for monitoring multiple interface-level events. In addition Vesper automatically conducts required testing and report signals correlated with these events based on analysis of test logs. The results of the experiments described in the paper indicate that Vesper's reports are free of false-negatives for signals that directly caused the event. Additionally, given minimal knowledge about the design, Vesper is able to accurately to select event-related signals out of the pool of signals two orders of magnitude larger with no more than 7 testbenches. The signal list reported by Vesper can be used by engineers to establish important principles of the design's functionality or can be incorporated into StressTest test generator for verification purposes.

One of the most important improvements to Vesper that can be done in the future is the removal of constant signals. Constant signals retain one value throughout the simulation, and thus the recorded value of these signals is always the same. So, during the analysis phase Vesper incorrectly concludes that these signals have high correlation with the event. Therefore, identifying these signals and removing

them from the final report will increase the precision of the tool. Another observation that we made was that some of the correctly reported signals were copies of each other on different levels of the design's hierarchy. A simple addition to the analyzing algorithm can observe that some signals have exactly the same value every time an event occurs and report only one of them. This will make the tool even more flexible and friendly to use.

Although Vesper started as a plug-in for StressTest, we also found that it might be beneficial to integrate Vesper with waveform viewers, like VirSim. In this application Vesper can produce a configuration file for the viewer, identifying small subset of signals that are related to the event and the interface signals triggering the event. This can help in both trying to understand the design's structure and debugging process, when the event of interest is caused by a bug.

The ultimate test for Vesper, however, would be running it in combination with StressTest on a buggy design. In that experiment Vesper would infer activity points from interface events and report these signals to StressTest to guide the stimulus generation. If a bug is found it would be reported to the user, otherwise the tests would run again with different events of interest. When trying to trace the bug the user might use Vesper again, with a different event specification to reduce a list of signals relevant to the bug occurrence in the waveform viewer. This approach would demonstrate that integrated debugging packages that include automatic stimulus generation tools like StressTest and profiling tools like Vesper can help engineers in conducting the verification process, aid them in understanding the design's structure, and increase the degree of automatization of the testing and verification of hardware.

#### 5. REFERENCES

- [1] Synopsis products: Vera, 2005. <http://www.synopsys.com/products/vera/vera.html>.
- [2] I. Wagner, V. Bertacco, and T. Austin. Stresstest: An automatic approach to test generation via activity monitors. In *DAC, Proceedings of Design Automation Conference*, 2005.
- [3] B. Wile, J. C. Goss, and W. Roesner. Comprehensive functional verification: The complete industry cycle.