Designer-Level Verification – an Industrial Experience Story

Stephen Bergman, Gabor Bobok, Walter Kowalski, Shlomit Koyfman, Shiri Moran, Ziv Nevo, Avigail Orni, Viresh Paruthi, Wolfgang Roesner, Gil Shurek, Vasantha Vuyyuru
IBM Corporation

***Abstract*—Designer-level verification (DLV) is now widely accepted as a necessary practice in the hardware industry. More than ever, logic designers are held responsible for the initial validation of modules they develop, before these are released to systematic verification. DLV requires specific tools and methods adapted for designers, who are not full-time verification experts. We present user experience stories and usage statistics, describing how DLV has been practiced in our company, using a dedicated tool developed for this purpose. A typical pattern that emerges is of designers devoting short, fragmented time periods to DLV work, interleaved with other logic development tasks. We observe that the deployed DLV tool supports this mode of work, since it is simple and intuitive. This demonstrates that a suitable tool can help DLV become an integral part of a logic design project.**

I. DLV Using a Dedicated Tool

Designer-level verification (DLV) is commonly accepted as a necessary practice in hardware design projects, for reducing the cost of bugs. Increasingly, logic designers perform initial validation of their code before handing it off to verification teams. DLV mainly includes observation of design behavior under mainline scenarios and selected corner cases, possibly encompassing assertion validation and lightweight bug hunting. Designers may encounter obstacles, due to their fragmented DLV work periods and their lack of verification expertise. In IBM this led to development of a dedicated tool, called the Debug and Verification Tool for Designers (DIVER)[1].

DIVER is based on *scenarios* defined in an augmented timing-diagram language, which allows assigning values to inputs at specific points in time, and defining repetitions, delays, triggers, and randomness. DIVER's main window is split into two parts (see Figure 1). The upper part is where scenarios are defined, in a spreadsheet-like table. The bottom part is a standard waveform viewer. The typical usage flow begins with editing a scenario in the upper pane. This may include concrete or random input stimuli, as well as expected values of outputs. The next step is simulating for some number of cycles, and finally viewing the simulation trace in the lower pane, with annotation showing mismatches of expected results. The user may also simulate in *run-to-failure* mode, which searches for assertion failures or unexpected results. The basic flow performs straightforward simulation, but some features utilize a formal verification engine.

[1]More details on the tool and user experiences can be found at https://www.research.ibm.com/haifa/papers/diver_DATE_2015_full_version.pdf

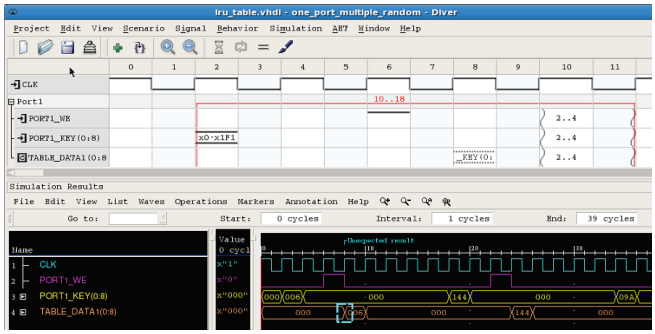

Fig. 1. DIVER's main window.

II. User Experience Examples

We present the first experiences of three logic designers using DIVER for their designer-level verification. Their main use of the tool was for testing newly developed logic, creating and running scenarios for each functionality they coded. Later, these scenarios were periodically run as regression tests.

A. User A: writing and reading data registers

User A developed a module that maintains internal data registers, and handles read and write requests to the registers.

1) Testing read requests: User A incrementally created a single DIVER scenario for checking all of the read functionality. A fragment of the final scenario is shown in Figure 2. First, he created a scenario for a single read request. In the initial cycle, specific data values are assigned to the internal registers. In column 5, a read request is driven by setting specific input values. Column 15 specifies an expected result check on the data output signal. User A gradually added more requests to the same scenario, first testing multiple requests with gaps between them, and then consecutive, pipelined requests.

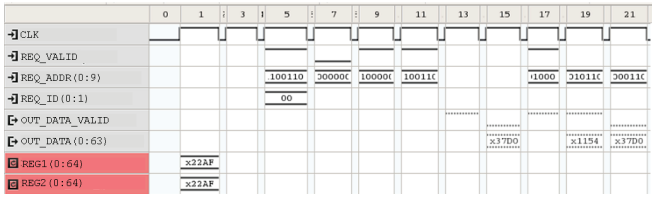

Fig. 2. User A scenario for checking pipelined read requests

User A progressively added sections to the scenario, each testing a newly implemented feature. Thus the single scenario served as a suite of tests, running together in one simulation. User A could view simulation results as a single waveform, with markers annotating mismatches of expected results.

2) Testing write requests: The scenario for testing write requests uses random values to drive the request input and initialize the data registers. User A wrote RTL assertions to check the data register values after each write request, and viewed assertion fails as markers annotating the waveform.

This style of testing, with random inputs and global assertions, allowed User A to use the run-to-failure feature, to repeatedly simulate the scenario with different random choices. When run-to-failure hit an assertion failure and produced a fail trace, it indicated either a logic bug or an incorrect assertion.

B. User B: an arbiter module

User B developed a module that performs arbitration between several requesters.

User B created a single scenario to check all aspects of arbiter functionality. He started by creating a very general scenario, with random values for many of the inputs. He then iteratively fine-tuned the scenario to describe only legal input patterns. In each iteration, User B simulated the scenario and received a fail trace, which either led to a bug fix in the logic, or to refinement of the scenario to eliminate an illegal input pattern. The scenario contains repetitions, delays, and triggers for checking several requests in sequence. To check arbiter fairness, User B viewed the simulation waveforms, and observed the even distribution of grants among requesters.

C. User C: an embedded processor

User C is actually a team of three designers, who developed new functionality in an embedded processor.

User C used DIVER to create a suite of tests covering every instruction in the processor's set. For each instruction category (e.g., all flavors of "Add"), they created a scenario that drives the various opcodes in the category, and includes expected-result checks on register values and internal processor states.

User C wrote a script that generated a textual scenario file for each instruction category. These auto-generated scenarios were then modified using the DIVER graphical interface, adding non-standard instructions from the same category. User C simulated this set of several tens of scenarios by a single call to DIVER's batch mode from the command line, which completed in less than 1 minute. This was repeated every time the design changed, to ensure that all scenarios still passed.

D. Insights from user experience

In these examples, we see the tight interleaving of verification work and logic development. Users created meaningful tests in short verification sessions, incrementally enhancing them in later periods. This mode of work requires a simple tool with a minimal learning curve. We see that users progressed from basic, isolated scenarios to more structured testing, ultimately creating small test suites and assertions that would accompany the logic in further project stages.

III. FIELD USAGE STATISTICS

In this section we present field data statistics and usage patterns, demonstrating how designers in IBM use DIVER. Data in this section was collected during a test period of 5 consecutive months. We only consider designers who ran simulations in at least 5 distinct days during this period. This amounts to several dozen users. Statistics for several usage aspects are given in Table I. A detailed analysis follows.

Usage Aspect	*Average*	*Median*	*Minimum*	*Maximum*
Time to first simulation	18:09 min	08:34 min	00:19 min	1:34:27 hrs
Average simulation time	24 sec	5.46 sec	0.86 sec	194 sec
Simulations	295.3	151.5	20	3100
Compilations	94.8	60.5	11	529

TABLE I. PER-USER DIVER-USAGE STATISTICS.

First, we check our claim that DIVER is an intuitive tool with a low adoption barrier. How long does it take then for a new user to get to the first simulation? For each user we measured the elapsed time from the moment he/she first opened DIVER until first clicking the Simulate button. We found that more than half of the users spent less than 10 minutes with the tool before simulating. This includes setup, compilation and defining a basic scenario. 84% of the users first simulated within 30 minutes, which well supports our intuitiveness claim.

We also claim short simulation times (from clicking the Simulate button until getting a trace). For each user we measured average simulation time, taken over all user's simulations during the test period. 61% of users got their traces in less than 10 seconds on average. 7% of users waited more than 2 minutes on average, probably simulating very long scenarios.

Finally, we check our assumption regarding short and fragmented verification efforts. We plotted the number of daily DIVER simulations of 5 users on each day of the test period (Figure 3). Each color represents a different user. Indeed, we can see a typical pattern consisting of several-days-long bursts of simulations, separated by longer intervals with no usage.

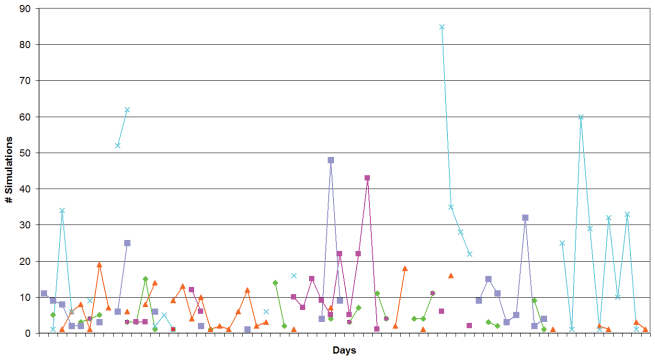

Fig. 3. Simulations per day of 5 users during 5 months.

From the data points mentioned above, we conclude that the tool is simple to learn and use. The usage patterns match our design assumptions: users simulate short scenarios, with a high frequency of code changes between runs, within brief usage periods interleaved with the development of the logic.