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. INTRODUCTION

Designer-level verification (DLV) [1] is increasingly gaining acceptance as a necessary practice in hardware design projects. Early identification and correction, by the developer, at the lowest level of integration is probably the most effective method for reducing the cost of bugs [2]. Unexpected behavior that is identified in the designer’s environment, before releasing the code to integration, is simpler to analyze, has no impact on the verification process, can be corrected without contributing to the logical-physical closure “churn”, and is therefore less likely to spawn new functional bugs.

In software projects, the importance of unit testing has long been widely accepted [3]. In hardware projects, several conceptual objections to designer-level verification have existed in the past. The first is the concept of strict separation between design and verification teams. The concern is that verification, if influenced too strongly by the point of view of the designer, might repeat the designer’s own errors. The second objection is that devoting designers’ time to verification is not a good utilization of their valuable expertise. However, the practice of DLV does not eliminate the design/verification separation. Designers still hand-off their code to separate verification teams, for systematic large-scale validation of the integrated design. DLV only improves the stability and functional correctness of this code. The time spent by the designer on DLV is easily offset by the reduction in failure-handling later in the project. Hence, more and more logic designers are involved in DLV.

Several activities can be considered DLV-related.

Demonstration and observation: This is probably the main DLV activity, where design behavior is validated under selected prime scenarios, aiming to expose fundamental flaws, and in addition a close look at selected corner cases.

Validation of in-code assertions and coverage points: Assisting the designer to answer the following questions: do my assertions correctly capture the situations I had in mind? Are my coverage points hittable?

Lightweight bug-hunting: Developing scenario templates, then massively executing random tests derived from these templates; alternatively, formally verifying in-code assertions as well as externally specified checkers.

Pre-release regression: Running accumulated test setups before any new release of the code.

Debug and correction process: Root-causing failures found by the verification team, comparing alternative solutions, and validating the chosen fix.

Code reviews: with leaders and peers.

Static code analysis: Using general-purpose tools, such as linters, to detect common coding errors.

The methods and tools currently used for DLV are diverse, usually based on simplified verification methods, such as simulation testbenches or formal verification (FV). However, in trying to apply standard verification tools to DLV, designers encounter several obstacles. The first is the need to interleave this activity with their design work, which typically results in short, fragmented bursts of DLV. Secondly, designers lack specific expertise in verification, which is a field with its own tools, languages and methodologies. Thirdly, verification tools are usually intended for use on large, integrated design units, and are not adapted for the small component level of DLV.

Several designated tools have been proposed for DLV. An early tool was TIMEDIAG [4], which provided a graphical waveform-style interface for defining simulation stimuli, and generated testbenches to be run (non-interactively) in a separate simulator. PathFinder [5], was based on an FV engine, which produced output traces that satisfied constraints, specified graphically by the user. Jasper’s Visualize tool [6], [7] also produces traces based on specified constraints, using an intuitive waveform-style interface. These constraint-based tools are useful for basic design exploration and assertion checking. But defining constraints is often unintuitive, in contrast to directly defining the behavior of inputs. For straightforward test cases, the exhaustiveness of FV is not necessary, and may require unnecessary computation resources. Other related tools are graphical waveform editors, provided by simulator vendors for simple stimulus definition [8], [9].
The contribution of this paper is in describing the successful deployment of DLV in IBM using an internally developed tool. We detail the considerations we had in designing the tool, and justify them based on user-experience stories and usage statistics.

II. THE NEED FOR A DEDICATED DLV TOOL

A. Past DLV practices in IBM

In IBM’s high-end server design teams, logic designers were required to specify correctness assertions to the code, and to define block-level\(^1\) coverage goals [10]. In some projects, designers were also required to run regression tests, prepared by the verification team, as a precondition for code release. Many logic designers exercised prime scenarios of their code and some targeted corner cases. This was done using simplified simulation environments. Simulation was usually carried out at the block level, yet some design teams ran scenarios at the unit level. A few designers also used FV. The time invested in these activities varied, by developer and team, from a few hours to a few days per module. The activity was not mandatory, was not systematically applied to all modules and was not consistently iterated with new code releases.

In contrast to the above, designers did invest a significant portion of their time dealing with functional failures. Detailed tracking and analysis of a design activity in IBM indicated that 37% of the designers’ time was dedicated to activities related to such bugs. Designers mostly dealt with implications of failures to modules they develop. This includes reproduction of the failure, root-cause analysis and correction.

This led to the decision to develop a dedicated tool for designer-level verification and debug.

B. High-level requirements

We identified the following three high-level requirements, which are necessary for a designer verification tool.

1) Low adoption barrier: Considering the short and fragmented effort that a designer can devote to DLV, the tool must be highly accessible and easy to learn. The tool should incorporate familiar components such as the commonly-used wave viewer and RTL compiler. The tool should hide any complex configurability in the designated verification tools, exposing only DLV-essential features. Finally, users should not be required to learn a new language, which would present a barrier to initial use of the tool.

2) Linear ROI: Ideally, the more time a designer spends with the tool, the more benefit she should gain from it. A designer should get a meaningful simulation trace of her design even if she can only spend 15 minutes with the tool. If more time is available, the designer may progress to inspecting corner cases or lightweight bug hunting. Therefore, in addition to quick test case specification, the tool must also support more complex stimuli creation and more exhaustive checks. It should be easy for the user to progress in both dimensions.

3) One-stop shop: Supporting all designer verification activities inside one single tool. These include compilation, creating testcases, simulating, and viewing results.

Based on these requirements, we developed a tool, called the Debug and Verification Tool for Designers (DIVER).

III. DIVER: A DESIGNER VERIFICATION TOOL

A. Concept

At the heart of our concept for DIVER lie scenarios. Scenarios specify stimuli to the design as well as expected results and constraints. Scenarios are defined using an augmented timing-diagram language, which at the very basic level allows assigning values to inputs at specific points in time. This simulation look and feel makes the working environment intuitive to designers (as opposed to defining input constraints) and lowers the adoption barrier. On top of simply assigning values to signals, the scenario language also allows defining repetitions, random delays, triggers and randomness.

Since the scenario language is more feature-rich than timing diagrams, we decided to visually split scenario definition from simulation results. Hence, DIVER’s main window is split into two parts (see Figure 1). The upper part is where scenarios are defined, and where complex constructs can appear. The bottom part is a typical waveform viewer, showing simulation results.

The envisioned usage flow is as follows.

1) Attach design, compile/load
2) Specify scenario in the upper pane

\(^1\)We use the term block to denote a small component written by a single designer. A unit is a larger component composed of several blocks.
Variation 1 Specify concrete input behavior (one value per signal at each cycle)

Variation 2 Specify predefined input patterns (clocks, counters, shifters, ...)

Variation 3 Add randomness (biased random values, random delays) and reactiveness (triggers)

Variation 4 Specify additional constraints

Variation 5 Specify in-model assertions, external checkers, coverage events

3) Simulate X cycles

4) View/query simulation results in the lower part of the screen and continue with one of the following activities:
   • Lengthen simulation with additional Y cycles OR
   • Resimulate (with a new random seed) OR
   • Refine scenario specification and resimulate OR
   • Modify design, recompile and resimulate.

The most basic usage flow is performing straightforward simulation, with input behaviors driven deterministically (Variations 1,2). Designers naturally think in terms of driving inputs in order to see specific outputs. Variations 3-5 are extensions of the basic usage and may provide users with more powerful results, thus, obtaining linear ROI. Moreover, introducing constraints, assertions and coverage opens the gate for applying formal analysis, which can yield the following results.

- Timing diagrams that satisfy constraints. This way, scenarios can be defined in terms of internal events, without specifying the exact input sequence that is required to generate such events. There are many cases where this would be more natural to the user.
- Timing diagrams, where specific coverage events are hit.
- An exhaustive check of assertions under the randomness allowed by the given scenario.

DIVER offers seamless transition from simulation engine to formal analysis engine, as needed, and thus combines the high speed and capacity of simulation techniques with the power of formal analysis under the hood of a single front-end.

B. DIVER- Detailed description of selected features

1) Scenario editing: In the upper pane the user specifies the scenario as an augmented timing diagram. The scenario can be edited in a spreadsheet-like table, having standard type-in and copy/paste utilities as well as wave drawing features using a pencil tool. These make the editing process easy and intuitive, especially for users who go only as far as Variation 1.

Several stimuli patterns match input behavior on many designs. Examples are clock, reset, constant, random and one-hot. DIVER provides the user with a rich set of such (parameterized) stimuli patterns, and allows the user to use them for driving specific inputs (Variation 2).

Even for explicitly-defined signals, there is no need to set values in each and every cell. Empty cells and undefined inputs receive a configurable default specification.

2) Outputs and internal signals: The desired values of outputs and internal signals at specific points in time can be described just like input stimuli. However, such values may serve three different goals based on user’s choice. The specification waveform may override the existing logic, it may define an expected result to be matched to the actual simulation result, or it may constrain a scenario, limiting result waves to only those which exhibit the specified values (Variation 4).

3) Repetitions and Delays: Hardware designs are reactive systems. Hence, stimuli patterns are likely to repeat over and over (with new random data drawn each time). DIVER allows defining such repetitions. The number of repetitions can be either fixed, unbounded or random within a specified range.

Delays allow the introduction of variable-sized spaces between specified events. This is useful when we want to see the design’s response to events in various orders and timings (Variation 3). Such delays will often be a part of a repeated pattern (see Figure 1).

Alternatively, delays can be specified to last until some condition is met. This condition, specified as a Boolean expression over design signals, serves as a trigger: events following a conditional delay will only occur after this trigger is activated.

4) Backend: DIVER interfaces with IBM's standard tools for compilation, simulation, FV and wave viewing. When the designer compiles her code, DIVER runs an HDL compiler to obtain a model. The tool then analyzes the model to get signal names, assertions and coverage goals.

When the designer asks to simulate a scenario, the tool generates an execution environment (testbench) which suits both simulation and formal engines. Depending on the existence of constraints and considering the design size, an appropriate engine will be chosen for getting a result timing diagram. In addition, the user may choose to simulate in a run-to-failure mode, instructing the tool to provide a timing diagram where an assertion fails or an unexpected results appears. A formal engine may perform an exhaustive check to find such failures, while a simulation engine may run multiple times, using different random seeds. Here too, the transition from simulation to formal analysis and vice versa is seamless.

After a simulation or formal engine produces a trace, DIVER checks for any mismatch of expected results, violation of assertions and reaching coverage goals, then annotates the trace accordingly, and displays the result waveform.

5) Supporting Debug Activity: DIVER helps designers to concentrate their debugging effort on their small blocks by projecting a failure trace from higher design hierarchies into a block-level scenario. This scenario contains only signals from the block interface and can be used for reproducing the problem, performing root-cause analysis and what-if analysis, then validating a fix and running regressions.

IV. User Experience Examples

We present the first experiences of three logic designers in using DIVER for their designer-level verification. The main
activity these designers performed with the tool was testing new logic during its development. Throughout the RTL coding of each new functionality, the designers created and ran scenarios (tests) for checking that function. After the function coding was complete, these scenarios were used as regression tests, which were run periodically to check that subsequent logic changes had not broken the existing functionality. We describe how each designer performed these main activities, and some secondary activities.

A. User A: writing and reading data registers

User A developed a module that maintains internal data registers. He implemented the mechanism for writing to the registers, based on input data and accompanying parameters, and for handling requests to read data from the registers.

1) Testing read requests: User A tested read requests independently of the write mechanism. He created a single DIVER scenario for checking all of the read functionality, beginning with a simple scenario and incrementally adding complexity. A fragment of the final scenario is shown in Figure 2.

First, User A created a scenario for a single read request. In the initial cycle, specific data values are assigned to the internal registers (REG1, REG2, and others). In column 5, a read request is driven by setting specific input values (REQ_ADDR, REQ_ID) and raising the REQ_VALID signal. Column 15 specifies an expected result check on the OUT_DATA signal.

After User A simulated and debugged this single request, he added more requests in later cycles of the same scenario. The requests had gaps between them, to allow each request to be handled completely before the next request arrived. When isolated requests had been tested and debugged, User A changed the scenario to include several consecutive requests without gaps between them. This tested that the logic could handle multiple pipelined requests.

2) Testing write requests: For the write functionality User A also incrementally created a single scenario, which ultimately served as a regression suite. In contrast to the scenario for read requests, which used specific values for all inputs, the scenario for writes uses random values to drive the request input, and the internal data registers are initialized to random values. Instead of specifying expected results, User A wrote inlined assertions in his RTL code, to check that the data registers contained the correct data value after the write. User A simulated the scenario using the graphical interface, and viewed assertion fails as markers annotating the waveform. We note that instead of RTL assertions, the same global checking can be done using expected results in the graphical scenario, with expressions that refer to input values at earlier cycles.

3) Bug hunting and assertion debug: The style of testing used for write requests, with random inputs and global assertions, is more general than the concrete scenario style, and supports more exhaustive bug-hunting. User A used 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 bug in the logic or incorrect coding of an assertion.

4) Design exploration: User A used DIVER when he became responsible for a piece of logic written by a different designer. He ran DIVER simulations to explore the design and study its response to different input patterns.

B. User B: an arbiter module

User B developed a module that contains two 3-to-1 arbiters. It receives requests from six requesters, and outputs up to two grants at a time.

1) Testing arbitration: Like the previous user, User B created a single scenario to check all aspects of arbiter functionality. However, there are some differences in scenario style.

User B 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. A fragment of the final scenario is shown in Figure 3. It contains repetitions for checking several requests in sequence, and conditional delays for triggering behavior in reaction to other events in the scenario. The scenario includes expected results to check that error signals never rise. To check the fairness of the arbiter, User B viewed the simulation waveforms, and visually confirmed that the distribution of grants among the requesters was even. To test different arbiter modes, User B manually changes some of the lines in the scenario and resimulates.

2) Design observation for power-optimization planning: User B coded a specific event as a coverage statement, then ran simulations and viewed the number of event hits reported by DIVER. He concluded that the event was too rare, and did not justify adding event-specific logic for power optimization.
C. User C: an embedded processor

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

1) Testing all instructions: User C used DIVER to create a suite of tests to verify a good machine path for every instruction in the processor’s set. They created a scenario for each instruction category (e.g., all flavors of “Add”). Each scenario is basically a small program that drives the various opcodes in the category. The scenarios include expected-result checks on register values and internal processor states at specific cycles.

For creating this set of several tens of scenarios, User C wrote a script that generated a textual scenario file (in a format used by DIVER) for each of the instruction categories. These auto-generated scenarios, which only drove a single instruction each, were then modified using the DIVER graphical interface, adding other instructions from the same category in later cycles. This was done using copy-paste of graphical scenario segments (similar to copy-paste in a spreadsheet). User C simulated the set of scenarios by a single call to DIVER’s batch mode from the command line. The batch run completed in less than 1 minute, and was repeated every time the design changed, to ensure that all golden scenarios still passed.

2) Testing subcomponents: User C reported that testing subcomponents, even before the entire module was complete, became possible with DIVER (as opposed to other verification environments), since DIVER allowed them to drive stimulus at any level, not only at primary inputs.

D. Insights from user experience

In all of the examples, we see the tight interleaving of verification work and logic development, with users continually switching back and forth between these tasks (also demonstrated in Figure 6). Users were able to create meaningful tests in short verification sessions, and to incrementally enhance the tests in later periods. This mode of work requires a simple tool with a minimal learning curve, and without extensive setup time. We also 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. Thus, the tool was able to lead them into more complex verification tasks. Finally we observe that each user performed a variety of DLV activities, demonstrating the need for a “one-stop shop” tool.

We noted that users chose not to use some advanced features of the tool, even when these were appropriate for the task at hand. Designers, within their tight schedules, cannot devote significant time to learning new features. This imposes a barrier to using advanced features, and emphasizes the importance of keeping the tool design intuitive and accessible in future developments. We also identified a necessary area for development: creating and managing variants of a scenario, which share most of their behavior but differ in some details. The challenge is to better support this in the graphical interface, without resorting to textual files or command-line mode.

V. Field Usage Statistics

In this section we present field data statistics and usage patterns, demonstrating how designers in IBM use DIVER in practice. DIVER sends basic usage information to a central database, including GUI invocations, simulations and compilations. All data in this section was collected during a test period of 5 consecutive months. We only consider the set of designers who ran simulations in at least 5 distinct days during this 5-month period. This includes several dozen users. We consider the usage of all other users to be incidental. Statistics for several usage aspects are given in Table I. We now take a closer look at some of these aspects.

<table>
<thead>
<tr>
<th>Usage Aspect</th>
<th>Average</th>
<th>Median</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to first simulation</td>
<td>18:09 min</td>
<td>08:34 min</td>
<td>00:19 min</td>
<td>1:34:27 hrs</td>
</tr>
<tr>
<td>Average simulation time</td>
<td>24 sec</td>
<td>5.46 sec</td>
<td>0.86 sec</td>
<td>295.3 sec</td>
</tr>
<tr>
<td>Simulations</td>
<td>695.3</td>
<td>151.3</td>
<td>40</td>
<td>3100</td>
</tr>
<tr>
<td>Compilations</td>
<td>94.8</td>
<td>60.5</td>
<td>11</td>
<td>529</td>
</tr>
<tr>
<td>Simulations/compilations</td>
<td>2.88</td>
<td>2.21</td>
<td>1.11</td>
<td>12.85</td>
</tr>
</tbody>
</table>

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 brand new user to get to her first simulation? For each one of the “non-incidental” users we measured the elapsed time from the moment he/she first opened the user interface until clicking the Simulate button for the first time. The distribution of users by time-to-first-simulation is shown in Figure 4. More than half of the users spent less than 10 minutes with the tool before they started simulating. This includes setup time, compilation...
time and the time it takes to define a basic scenario. 84% of the users needed no more than 30 minutes to get to their first simulation, which well supports our intuitiveness claim.

We also claim short simulation times, meaning a short time period from the moment the user clicks the Simulate button until she gets a result trace. Note that the simulator runtime is only a part of this time period. DIVER also has to translate the scenario into a testbench and to analyze the trace for violations of expected results or failing assertions. For each user we measured average simulation time, taken over all simulations run by the user during the test period, excluding simulations that require formal analysis. 61% of users got their traces in less than 10 seconds on average. 7% of users had to wait more than 2 minutes on average. They are probably simulating large units or very long scenarios.

Another interesting data point is how many simulations users actually ran during the test period. This number seems to widely vary. Some users only had a few tens of simulation over the 5-month period, most users had a few hundreds of simulations and some users used DIVER to obtain more than a thousand simulation traces.

We now want to somewhat justify our requirement for a “one-stop shop” tool. DIVER allows the user to compile and to simulate in the same window. We can see why this is important if we count the number of compilations each user had during the 5-month period, and compare it to the number of simulations. We found that most users recompile their design every 1–6 simulations on average, which is a surprisingly low simulations/compilations ratio (this ratio probably grows as the design becomes more mature). The interleaving between compilations and simulations is also demonstrated in Figure 5, where we focus on one user using DIVER during one day. Each bar represents either a simulation (yellow), successful compilation (blue) or an unsuccessful compilation (purple). Compilations fail when the user has a syntax error for example.

![Fig. 5. Usage of one user during one day. X-axis (time) is not to scale.](image)

Finally, we check our assumption regarding short and fragmented verification efforts. We chose five users and plotted the number of daily simulations each user ran with DIVER on each day of the 5-month period. This is shown in Figure 6. 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. 6. Simulations per day of 5 users during 5 months.](image)

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.

VI. CONCLUSIONS

Early verification by logic designers is essential to improve the quality of modules released to systematic verification. However, most verification tools are not well adapted to the needs and work patterns of designers. In this paper we demonstrated through experience stories and usage statistics, how a dedicated designer-level verification tool (DIVER) can help designers verify their logic more robustly.

The presented usage statistics substantiate our assumption on the short, fragmented time periods designers can devote to DLV. Statistics also show how a highly intuitive graphical tool can overcome this limit, and provide designers with meaningful verification results in as little as 10 minutes. From the user stories we learn how an appropriately designed tool can lead designers all the way from simple deterministic test-cases up to lightweight bug-hunting effort and to the development of pre-release regression suites. We also observe the importance of a single tool, supporting multiple DLV activities.

Integrating DLV into the logic design cycle is a significant step towards closing the verification gap. We conclude that with the support of a dedicated tool, this can become a reality.

REFERENCES


