Integration of modern verification methodologies in a TCL test framework

Combining the "quick turnaround" development process of scripting languages with UVM-inspired techniques to reduce test creation and update times in a hardware/software co-design environment.

Matteo De Luigi, Consulting Engineer, Ogheri Consulting GmbH, Munich, Germany, (*matteo.deluigi@alessandroogheri.com*) Tel +393279547290

Alessandro Ogheri, Consulting Engineer, Ogheri Consulting GmbH, Munich, Germany, (*ogheri@alessandroogheri.com*) Tel +491721404451

***Abstract*—Hardware and software verification are, taken alone, challenging enough a task. When the necessity arises to carry both out at the same time, the problem is compounded. Tight deadlines coupled with frequent updates in specification and implementation make it vital that verification engineers adapt swiftly to said changes. In the specific case we are going to discuss the device under test consists of two hardware modules and a set of firmware packages for one of them. Each component is developed by a separate group, and therefore subject to change with different, non-overlapping schedules. In this paper we present a firmware verification environment which takes concepts from the state of the art in digital hardware verification techniques and from the rapid prototyping approach of the software world, combines them, and applies the result to the verification of firmware routines with a limited set of possible input parameters but very long run times. The resulting environment, optimized for developer efficiency, enables the creation of tests in a fraction of the time it would take with a more classic approach.**

***Keywords*—TCL; simulator; UVM; verification environment; run time**

I. INTRODUCTION

Plenty of books and articles about verification start by pointing out the increased portion of resources the subject is taking up in contemporary development teams [1]. The field is mature enough to have developed methodologies and tools that fit the commonly recurring use cases. Occasionally, the constraints, requirements or project structuring deviate from the most common case and a verification engineer has to analyze the situation and come up with a solution which, while taking inspiration from the established methodologies, needs to be tailored to the specific case at hand.

In this paper we present an environment and development flow we have devised to meet a customer's very specific needs of fast turnaround in a hardware/software co-design environment with frequently changing complex specifications.

II. AREA OF APPLICATION

The Device Under Test (DUT) is a microprocessor-driven controller for an analog module. The whole design is developed in-house by three cooperating teams: one for the analog part, one for the digital part, and one for the software that controls the digital part. The task we were assigned was to check a non-trivial, protocol-like flow of control signals between the digital controller and the analog part.

The algorithmic complexity of the checks were such that simple assertions would not have sufficed; worse, the functionality to be tested was located at the intersection of the three worlds (analog design, digital design, and firmware), which meant tracking three different specifications at the same time, each with its own changes and updates. Reaction time was therefore critical: during the early development stages, one test took eight hours to run to completion, consuming a whole working day for a single run. With an assigned time slot of one to two weeks to complete this task, had we used a traditional model, we would have been able to run the test only five to ten times before the deadline, all of this with the risk that bugs may have been found (so new reruns would be needed) or that any specification might change at any time.

In contrast to this complexity of the algorithmic evolution, the input space is well-behaved: parameters are few in number, and they are largely independent from each other. Randomization needs are extremely limited.

III. Analysis

In a previous project, at the customer's request, we checked the aforementioned algorithmically tricky part by developing an extension to the Constrained Random Verification (CRV) environment used for the functional verification of the digital hardware component of the DUT. The extension provided the ability to check the evolution of analog signals at the interface between the analog and digital design, interaction which does not fit the model of the existing HDL-based firmware verification environment.

When a port of the old environment was needed for a similar, but more complex project, we have carried out an analysis of the approach taken in the previous one.

Minimizing run time turned out not to be a top priority: regressions run in batch mode during the weekend, so, as long as the overall run time was kept within reasonable bounds, a little performance hit in batch mode was acceptable. Determining the optimal set of stimuli for reaching full coverage is not an issue, either: given the reduced size and structural simplicity of the input space, it is possible to achieve satisfactory use case coverage by explicit enumeration.

The critical quantity to optimize is development time. Tight deadlines, quick reaction to frequent updates, flexibility and resilience to changes in the design (our DUT has a non-negligible effect on the whole chip's performance, so any measurable improvement can be worth the hassle) require the development turnaround to be as fast as possible.

Committing to the same CRV-tool based approach would provide the following advantages: an already available verification component tested in a previous project, a powerful language which integrates with the simulator, native support for modern verification methodologies, and, finally, good portability across simulators: at one point our design had to be ported from the simulator it was originally developed on to one from a different vendor, so sudden platform switches are not that far-fetched a prospect.

Our experience also brought to light the following disadvantages: slow development turnaround (each interactive restart of a simulation took up a non-negligible amount of time), the proliferation of separate sets of stimuli and regression environments that need to be maintained (two just for firmware verification: CRV-based and HDL-based), the need for specialized skills (not only for developing the environment, but also for applying stimuli; nobody else in the firmware verification team had the required skills), reliance on a complex metaprogramming step (a large part of the data and logic that controlled checks had to be extracted through text processing methods, which required either resorting to an external scripting language, or performing it within the CRV-tool, which it was ill-suited for the task).

Most of the above drawbacks are known and well accepted by hardware verification engineers, because, in exchange, one benefits from advanced constrained randomization, automated coverage data collection, and corner-case stimulation through the generation of input sequences that normally would not be tried. Our project, however, needed none of these: randomization requirements are trivial and some tests are too long to be run in sequence with any other (exhaustive checks of initial and final states are performed instead), which meant that none of the CRV-specific features would be used. To summarize: the CRV-based tool served its purpose, but did not fit the nature of the problem the customer was trying to solve.

The most evident drawback was development turnaround time: some details of the algorithms to verify were full of special cases (many of which were not machine-extractable), so the interactive debugging time needed to get everything “just right” dominated development time. CRV-based tools, in general, excel at long, unassisted runs where multiple scenarios are tried in complex sequences (which is not our use case), but do not perform as well in combination with a quick try-and-fix development style (each reload, even if just to fix a typo, would take about a minute).

An approach based on TCL [2], the de facto standard extension language for digital hardware simulators, has been considered as an alternative. It would bring many advantages: we still rely on a powerful, standard programming language which integrates with the simulator, but we would not need a separate regression environment (a breakpoint-based TCL system can run in parallel with our existing directed-test environment without disruption), and would experience lightning-fast load and initialization times (about one second). The single most important benefit, however, would be flexibility in development and quick turnaround times: test logic which is hard to get right on the first try can be prototyped within an OS shell. Once it works, the relevant part of the script can be plugged directly into the global test environment that runs on the simulator. As we will see later, such tests can also be performed using logged data from an actual simulation as input.

IV. CHALLENGES AND STRATEGY

What has shaped our environment the most were not the advantages it would have brought, but the challenges we had to face:

A. The existing, proven CRV-tool based environment we had developed, while using only a subset of the functionality provided by such a powerful tool, does not fit the programming model provided by a standard TCL environment.

B. Plain TCL provides no specialized constructs for verification, let alone modern verification methodologies

C. To fit into an existing firmware verification flow, the environment must be developed as an add-on for an existing verification environment: it must follow specific testbench interaction protocols and logging conventions, and have minimal impact on the current verification setup, designed around principles of simplicity and uniformity in behavior.

D. Interacting with simulated hardware is a non-standardized process in simulators (each vendor defines a different set of commands to do so), and is simply not supported in a plain TCL interpreter run from an OS shell.

The four challenges listed above have been solved by devising a four-layer approach as depicted in Figure 1, where each layer addresses one of them.

The bottommost layer abstracts away the TCL primitives peculiar to the simulator and provides a functionally equivalent, but simulator-independent, replacement set of primitives. If we conform to this API we can also write tools that mimic the behavior of a simulator based on, for instance, input log files produced from an actual simulation, so that the simulation can be "replayed" into the checker (provided the checker is purely passive).

A second layer will abstract away additional functionality specific to the environment which is not provided by the simulator itself: for instance, testbench logic for tallying error messages, advanced logging capabilities, etc.

The third layer will provide an infrastructure which enables an approach to verification inspired by the UVM (Universal Verification Methodology) [3]: events, simulation phases, and objects that can exchange data items. Once this layer is available, the proven test module from the previous project can be ported through a mechanical process of adaptation to the new infrastructure, thus becoming a member of the fourth layer: the application layer. Each layer has access to the API of any other layer at a lower abstraction level.

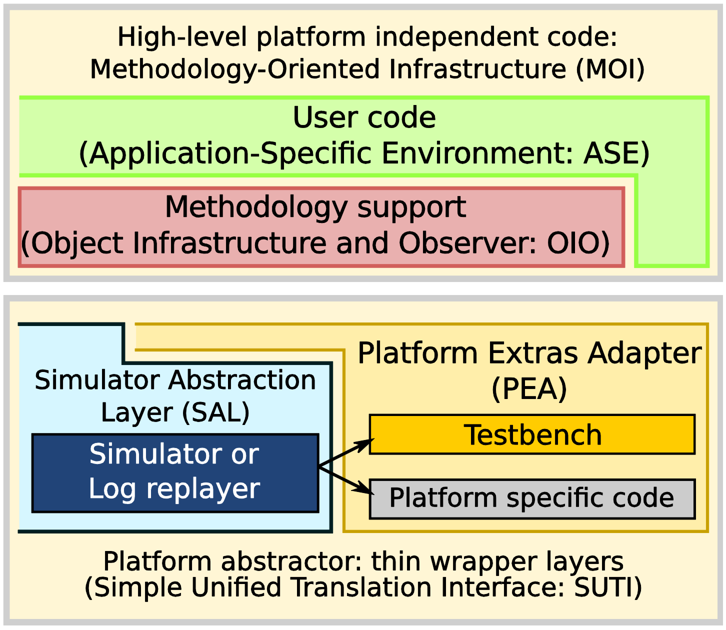
High-level platform independent code:
Methodology-Oriented Infrastructure (MOI)
User code
(Application-Specific Environment: ASE)
Methodology support
(Object Infrastructure and Observer: OIO)
Simulator Abstraction Layer (SAL)
Simulator or Log replayer
Platform Extras Adapter (PEA)
Testbench
Platform specific code
Platform abstractor: thin wrapper layers
(Simple Unified Translation Interface: SUTI)

Figure 1: general architecture of our TCL-based verification environment

The four layers are divided in two macro-groups: a low-level group, which deals with abstracting away the specifics of the platform, and a platform-independent high-level group, which implements the infrastructure and main functionality in a portable way.

V. DETAILED ARCHITECTURAL DESCRIPTION

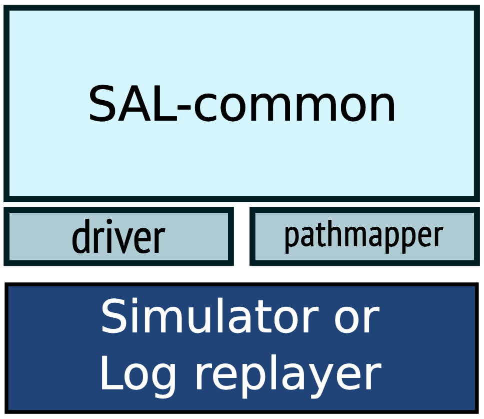

Figure 2: the structure of the Simulator Abstraction Layer. The driver and the pathmapper are the only components where simulator specific code is allowed

A. *Simulator Abstraction Layer (SAL)*

The purpose of the Simulator Abstraction Layer (Figure 2) is to provide access to all the needed functionality of a simulator and allow interfacing to simulator-like programs while, at the same time, minimizing porting effort.

There are four main challenges in creating a simulator-abstraction layer: design a runtime model, devise a unified set of commands and functions for a “virtual simulator”, provide a unified syntax for paths, and minimize porting effort to different back-ends.

The SAL tackles the first (and the relevant part of the third) challenge by exposing a single unified API to the verification environment, by implementing almost all of its own internal logic in a simulator-independent way (only about 300 lines of code at the time of writing), and by interacting with the simulator only through a tiny, simple “driver API” module, currently consisting of fifteen functions, most of them just one line in length (all they do, in most cases, is straightforward syntax translation of primitive operations like “register a callback on a signal change”).

The second challenge, i.e. abstracted access to hardware paths, is trickier to solve: syntax rules for paths vary dramatically across not only simulators and projects, but also Hardware Description Languages (HDL). To keep the implementation simple, we once again take inspiration from modern verification techniques and introduce a concept inspired by the portmapper of the eRM (e Reuse Methodology) and the “interface” construct of SystemVerilog [4]: the “pathmapper”.

The pathmapper is a small simulator and project dependent TCL module which SAL depends on. Its task is to translate the simulator, language, and project specific path names of interest into a standardized virtual representation which uses a slash as a path element separator. These new names, called “virtual paths”, need not be a direct translation of the physical paths into the new naming conventions: each verification environment is free to organize them as it sees fit, implementing the concept, present in modern verification methodologies, of having a native identifier abstract away the details of a hardware path. In our TCL implementation, the role of the pathmapper is to fill an associative array where each key is a string containing an absolute “virtual path name” in a special normalized form, and the associated value is the native path which will be substituted in back-end commands whenever a SAL operation is requested on the corresponding key. In traditional CRV-based tools, it is common to associate HDL signals to CRV-language identifiers by explicit enumeration in the verification environment's source, possibly by having the mapper generated by external ad hoc metaprogramming scripts. With TCL, due to its integration in simulators, it is possible to algorithmically discover the structure of the hardware at run time by using the simulator's path analysis functions. This means that the script can be adaptive to changes in signal names or missing/added signals (for instance, different chips in the same family having different numbers and types of peripherals)

To give an idea of the power of this approach, the prototype of our environment could be ported across two related but different projects by changing just one configuration line in the pathmapper (i.e., the native path which served as starting point for auto-discovery): once plugged in the new project, the pathmapper could figure out the names and locations of all the needed signals by algorithmically querying the design, even though the names of the individual signals were different in the two projects.

In summary, the SAL can be ported to a new simulator by adapting only two small files: the driver and the pathmapper. Porting to a new project running on the same simulator only requires changes to the pathmapper.

It is important to note that SAL is strictly a simulator-abstraction module. Any functionality not normally provided by a simulator, but implemented by platform-specific code (for instance, message verbosity levels or testbench-specific error tallying and statistics) is not the responsibility of this layer and must be implemented in a separate one: the Platform-Extras Adapter, or PEA.

B. *Platform Extras Adapter (PEA)*

The scope of the Platform-Extras Adapter is, as said earlier, to abstract everything which is platform-specific but not simulator-specific. In our project, PEA provides a configuration database, goes through the project-specific protocol for reporting DUT errors (for instance, interface to the error tallying code in the HDL testbench), and formats log messages according to specific guidelines.

Like SAL, PEA is divided into two blocks: a platform-independent main block (which interacts with the simulator through SAL) and a small platform-specific driver. Since abstraction of the simulator has already been taken care of by the SAL driver, the PEA driver will deal with abstracting only platform-specific but not simulator-specific functionality (for instance, the testbench).

In our current environment, a PEA driver needs to implement only five functions.

C. *Object Infrastructure and Observer (OIO)*

This layer provides simple object-like data structures which can hold high level representations of hardware interactions, thus enabling transaction-based verification. It furthermore provides a framework for “handlers” (roughly equivalent to the UVM's “component” and the eRM's “unit”), which can connect to each other and exchange transactions.

The operational model of the “handler” is based on the UVM's analysis port and its publisher/subscriber model: each handler, at environment initialization time, subscribes to a set of lower-abstraction-level transaction handlers, from which it receives transactions. These lower level handlers can perform checks and/or publish transactions for the benefit of higher level handlers.

The environment also provides handlers for the most basic transaction: a signal level transition. This kind of handler is called “porthandler” and is special: while providing exactly the same output interface as any other handler, it takes its input not from other handlers, but by registering a breakpoint-on-change callback for a signal. When the callback is triggered because the associated signal changes value, the porthandler publishes a data item which contains the signal's name, its new value, and the current simulation time. This makes porthandlers the primitives on which hierarchies of handlers are built.

The OIO also implements the concept of “phases”, inspired by the equivalent concept in the UVM. Our current implementation needs and implements only three: “init”, “run”, and “finalize”. More can be added in the future if the need arises.

The “init” and “finalize” phases involve no advancement of simulator time. What they do is trigger appropriate methods on the handler objects (phase_init and phase_finalize, respectively). The “run” phase (phase_run) encompasses the temporal duration of the simulation. The phase_run method of a porthandler is triggered as a breakpoint callback every time the signal it is associated to changes; other handlers' phase_run method is invoked every time one of the lower-level handlers they subscribe to publishes a data item.

D. *Application Specific Environment (ASE)*

This is the level of abstraction in which the tests (or application-level code) reside.

This level contains the equivalent of “environments” in functional verification languages. It can contain passive monitors and checkers and, if instantiating an active testbench, it can contain sequence generators and sequence drivers.

Once the three lower levels (SAL, OIO, ASE) have been implemented, we have recreated a verification infrastructure similar enough to our old CRV-based one that porting our checker module is now mostly a matter of mechanical translation. In order to gain the full power of the new environment, however, one element is still missing.

VI. A SECOND TARGET: NOSIM

We have added a small HDL data monitor to the HDL testbench for our DUT, which can, among other things, generate a log of all the changes in signal values with associated timestamp, in a manner similar to VCD files.

We have then developed a TCL script, called NoSim (short for “No Simulator”), whose purpose is to read in a signal transition log file previously produced by an actual simulation and pretend to be a simulator which is running the logged simulation: NoSim's API allows to set breakpoints on signal changes, and simulation time to be advanced (and queried). NoSim also keeps track of the current value of signals and can invoke registered callbacks when it reads “log events” that would trigger one. All of this in about 800 lines of TCL code.

After writing SAL and PEA drivers for NoSim, along with a pathmapper, we gained the ability to fire up a standalone TCL interpreter from a shell and run exactly the same TCL checker code we run in on top of a real simulator, provided that the test does not force or deposit signal values: we can rerun the same checker, or a modified (but still passive) checker, without rerunning the same HDL simulation.

VII. RESULTS

Let us consider the example in Figure 3, which reports run times (in hours) for some tests with three different setups. The “simulator quiet” profile refers to a traditional run on top of a simulator, with verbosity set to minimum to keep log files as small as possible and save postprocessing time. NoSim profiles refer to reruns of the same tests as standalone scripts using NoSim as back-end. The “NoSim quiet” profile uses exactly the same verbosity level as the simulation, while “NoSim verbose” has the verbosity level set to maximum. It can be seen from the graph that run time gains are significant. This approach also allows performing ad hoc procedures like processing input logs by hand to extract only “interesting” parts and reconfiguring the checker to concentrate on the reduced data set: in most practical cases we have encountered, processing and reconfiguring take about a minute, and rerunning a reduced log takes only a few seconds per iteration. This way it is possible to weed out many tricky details in a time which is shorter than the one required by a full simulator run by orders of magnitude.

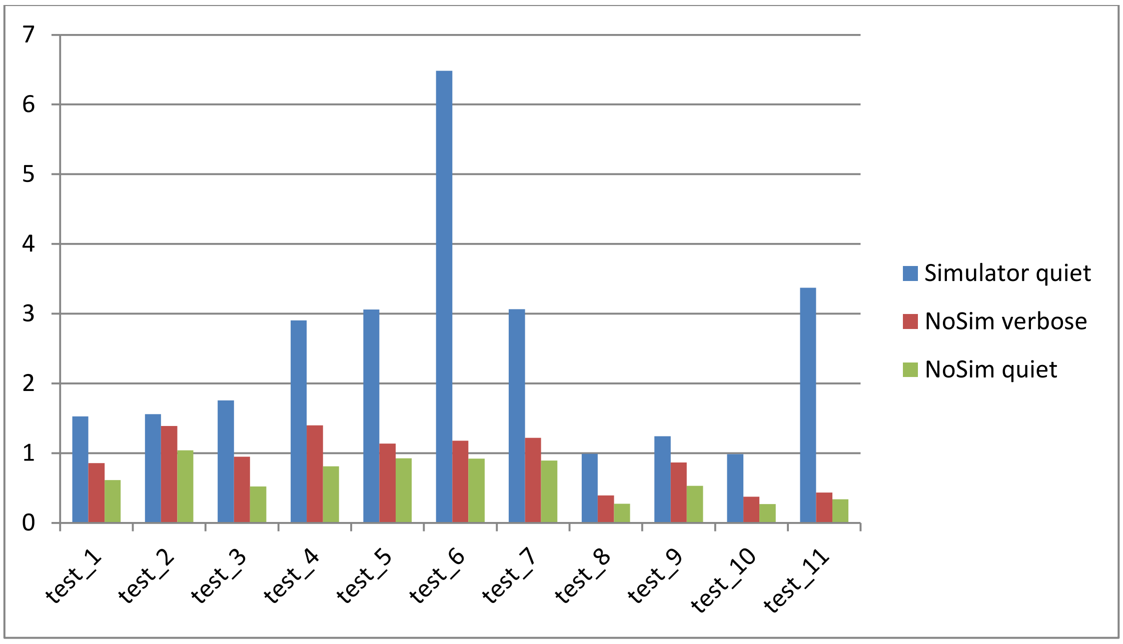

Figure 3: Comparison of run times (in hours) for some tests with different back-ends.

By developing the test code directly in TCL, we do not merely create quick prototypes: we quickly create the final test already.

VIII. Conclusion

Modern, advanced verification techniques implemented in a TCL-based environment reduced development turnaround and enabled complex tests as an add-on for a directed verification environment, requiring minimal changes and no disruption in the existing flow.

The environment infrastructure is portable across multiple back-ends, allowing offline usage (i.e., without a simulator), quick reaction to environment changes, easy deployment over a variety of projects, and code reuse. Offline usage is the main benefit here: after the addition of a log-replaying module, the environment (or parts thereof) can be invoked interactively from an OS shell.

The techniques described in this paper reduced the time needed to develop new tests and update existing ones in a project with very tight development time constraints, and permitted reuse of the same test code over multiple back-ends.

References

[1] S. Palnitkar, "Design Verification with e", Prentice Hall Professional, 2004.

[2] http://www.tcl.tk/

[3] K. A. Meade, S. Rosenberg, "A Practical Guide to Adopting the Universal Verification Methodology (UVM). ISBN 978-1-300-53593-5. Published, 2013.

[4] C. Spear, "System Verilog for Verification, A Guide to Learning the Testbench Language Features," Springer, 2008.