UVM-SV Feedback Loop – The Foundation of Self-Improving Testbenches

Andrei Vintila, Sergiu Duda
AMIQ Consulting
Bucharest, Romania
andrei.vintila@amiq.com sergiu.duda@amiq.com

***Abstract*-This paper is based on an article submitted to DVCon EU 2022 entitled "How creativity kills reuse - A modern take on UVM-SV TB architectures". The subject of the former paper is an architecture that allows the user to define a fully modular testbench (TB) that does not have hard coded constraints, components, or tests. The motivation of developing such a TB is to allow external control over the contents of an UVM-SV environment, as well as over the control variables and randomization constraints of the virtual/higher layer of sequences. One of the biggest advantages of such an architecture is the possibility to separate the stimuli generation from the SystemVerilog (SV) environment, opening the door to outside intelligent systems that can process data and improve the efficiency of stimuli generation. In this paper, such a system is presented, where data is gathered from simulation results and used to automatically adjust the input stimuli, in conjunction with other external variables.**

I. Introduction

In computer science, the most important actions undergone by any application are data processing and data transmission. Computing, networks, databases, analysis, artificial intelligence, and any other topic related to computer science is based on processing, storing, and moving data.

On an abstract level, hardware verification is no different since it is in fact part of the same spectrum. Stimuli are generated based on certain rules and patterns which are then processed by the TB to generate the expected outcome and in parallel processed by the design-under-test (DUT) to get the actual outcome. The comparison of the two outcomes is another form of data processing that results in computing the passing rate. Moreover, this flow is accompanied by the sampling of both input and output data which is another form of data processing that gives the coverage. This data is then moved throughout multiple components of the TB and ends being transmitted and processed once again by an external tool which gives a visual representation of this data in a human readable format.

Up to this point, everything can be optimized, and the approach allows a high degree of automation and abstraction. Once the results of a collection of tests have been acquired and can be read by the engineer, the process of achieving full passing rate and complete coverage begins. In most cases, this process is done manually and consists of multiple iterations of data analysis and incremental TB modifications. This is the main weakness of the verification process and the most time-consuming task.

However, this lowlight is not implied by the nature of the work. The data that is output by the verification process is machine-readable and many other forms of data can be generated by a TB through sampling and statistics gathering. Hence, the manual iterations of coverage analysis and incremental TB updates are a choice, not a requirement.

Steering away from this long-applied way of working would mean to replace the manual process of analyzing data and updating the testbench to developing intelligent algorithms that can predict the necessary adjustments to the input data based on the output data of the TB. This would still be an incremental process, but it has a much higher rate of automation, and it has the potential of removing a good chunk of the "grunt" work. For this technique to have an applicative potential, the most important requirement is to design a TB architecture that should allow external control over the stimuli generation and result collection. By having both of these data available outside the TB and outside the UVM-SV language scope, post-processing and automatic stimuli adjustments can be done through various algorithms depending on the project's nature. The objective is to establish a feedback loop.

II. Application and Prerequisites

The first step to create an automated way of improving stimulus based on simulation results is to have a TB architecture fit for the task at hand. Such an architecture implements a generalized way of controlling the creation of components as well as the randomization and execution of sequences. The SV language allows the user to dynamically assign certain variables at runtime by providing means of reading and writing files as well as passing flags and values through the command line via plusargs [1]. All these methods allow the user to employ any form of data processing that the operating system (OS) permits in order to optimize the TB input data.

The architecture presented in article [2] implements a system that allows the users to dynamically control the contents of the entire environment, allowing it to change from one simulation to the next one. On top of this, it allows queuing multiple UVM sequences from an external source and facilitates the creation of various test cases without having multiple hard coded uvm_test files (Figure 1). To solidify the previous statement, all the fields of the sequences can be controlled through the same means. This implies that any testcase can be created dynamically employing any number of applications, scripts, or algorithms that the OS allows.

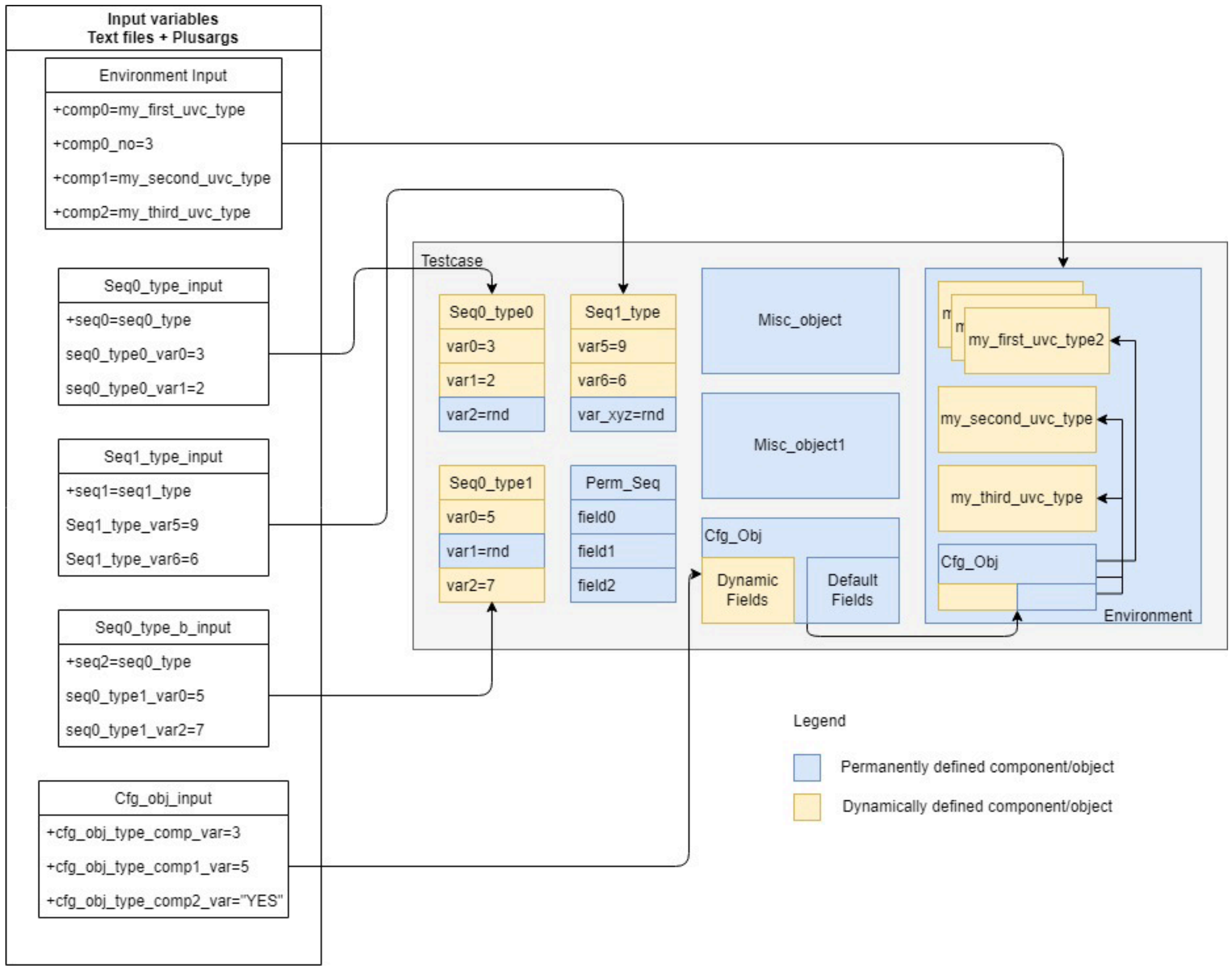

Figure 1. Dynamically controlled TB

This paper aims to instruct the reader step-by-step through the usage of the provided architecture with examples that create various TB topologies. Such topologies can either be top-level or block-level. Most of the necessary code is made available under an open-source Apache license. Moreover, a series of guidelines for deploying different use cases are provided.

The main deliverable is a library of objects that implement all the necessary methods to create dynamic testbenches. Moreover, a complete TB example is also included. In order to use it in a real/industry project just replace the basic UVM components and objects from which the TB classes are extended with their counterparts from the given library. Those classes implement additional methods that are not part of UVM and allow the dynamic control of variables and building of components/objects in an easy and seamless way. (Ref [2])

III. FEEDBACK LOOP CONCEPT

The concept of feedback loop implies diverting the output of a system towards the input. In a lot of software applications, this approach is typically used for error detection. By itself, verification is a form of loop, where the output of simulation, in the prospect of pass/fail and coverage results, drives the forward movement of the verification and design effort.

The traditional way of doing simulation-based verification assumes that the TB is a standalone entity that covers a certain verification space and validates the design against a golden model. The problem with this approach is that most of the time is spent on analyzing the same set of data that suffers minor updates. This approach is simpler, but it requires a large amount of repetitive work. A more complex, but automatic mode of achieving the same outcome is to establish a correlation between the results of a simulation, any other data that can be gathered, and the inputs that control the outcome. (Figure 2)

Any combination of system variables, statistical approximations, and simulation results can be used to generate the structure and control flags of a TB, as well as the stimuli generated inside the test sequences.

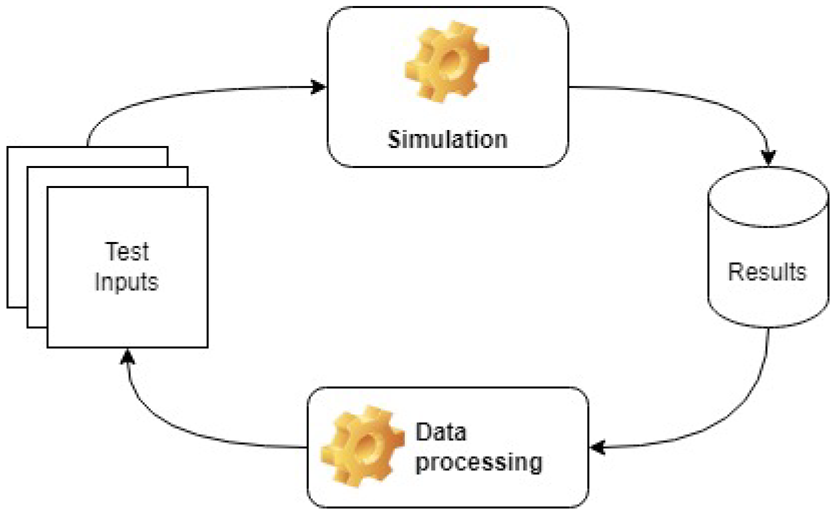

Figure 2. Feedback loop concept

This can prove harder in certain applications, but it is certainly a more refined approach. A simple way to find out if such an approach is useful is by approximating a transfer function for one's DUT. The requirement is that a certain output can be assumed for a known input, or a stochastic function can be described to act as the transfer function. If that requirement is met, then it is almost a guarantee that there is an algorithm that works better than manual analysis of the results. More so, the advantage of replacing the manual analysis is that a script thrives doing repetitive work contrary to a human. The same way the industry has moved away from hard coded test vectors, there comes a time where the amount of data and the complexity of the applications will bring the requirement of a higher layer of abstraction, one that implies mental prowess, rather than mechanical repetition.

Let's take a simple example: we have a simple system that applies a mathematical function, and we expect to cover a certain space for the inputs and the outputs. The purpose of this exercise is to see the steps that an engineer or a tool should take to solve the issue of coverage closure. The nature of randomization means that certain bins will be overhit while others will not be covered. The most time consuming and iterative process is to do manual adjustments to the constraints to improve the chances of filling the coverage holes. By taking the solution a step further we can properly optimize this process by trying to achieve complete coverage with a relatively smaller number of transactions. The purpose of such an exercise is to try to minimize the exploration of dead space. (Figure 3)

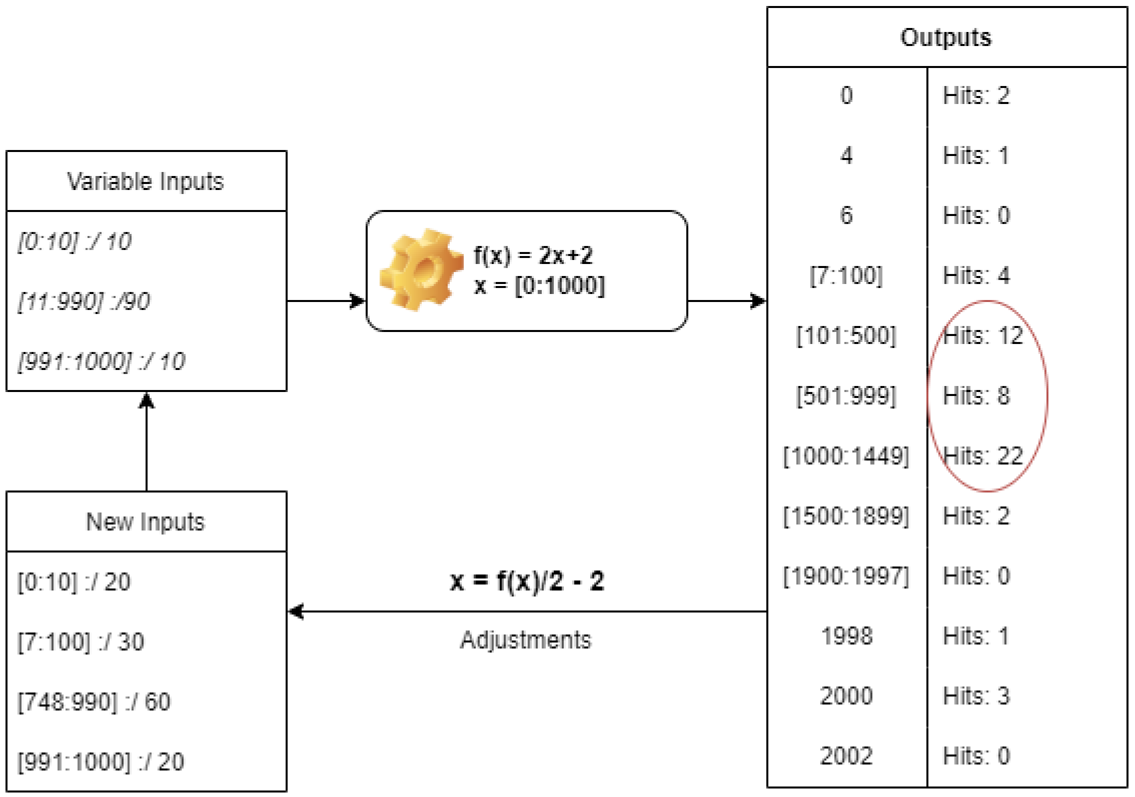

Figure 3. Coverage Driven Constraint Adjustment

The example in the above figure shows a correlation between a covered value (output) and a constraint (input). This can by itself is a system that can be optimized with a feedback loop. In traditional verification, that feedback loop is the engineer doing manual adjustments to the code.

In case of constraint random verification, the highest level of control we have are the constraints applied on virtual sequences. The functional goal of a UVM TB is to pull enough control knobs to the highest sequence abstraction layer to allow complete exploration of the verification space. During coverage closure, the role of the engineer is to figure out the relations between the coverage results and then adjust constraints to meet coverage goals. In most cases, the dependencies between coverage bins and constraints are not complicated and easily approximated, but a direct translation for uncovered bins is not desirable. You always want randomization to go slightly outside the scope of coverage.

So then, what are we trying to accomplish? Firstly, we want to free the engineer of doing boring and repetitive work. Secondly, we want to optimize the process of space exploration via randomization. Thirdly, we want to cover the last few difficult coverage holes in a timely manner. The steps to accomplish all these actions are the following:

- Adapt the TB in such a way as to be able to recursively adjust the TB configuration, sequences, constraints and even architecture without recompilation and in a compact way suited for automation
- Automatic coverage analysis and exposure of the bins and their hit rate in a machine-readable form
- The capacity of applying with ease any algorithm on the feedback loop between results and inputs. (For some problems a direct mathematical relation is possible, for more complex relations a self-learning or space exploration algorithm can be employed)

IV. Stimuli Abstraction, Higher level of control

Taking in consideration the experience we have on past trials, the nature of the algorithms or the automation that can be done for reaching coverage maximization is limited by the employed TB architecture. It is significantly more important to adapt the stimuli controls to the problem at hand than to try to adapt the problem to the stimuli.

A popular approach of employing external computation and automation within a UVM environment has been through DPI. However, the overhead on this approach is heavy and the limitations and complexity implicate a lengthier process than manually adjusting controls based on result analysis. All projects that we've seen applying ML algorithms for coverage maximizations were not scalable due to these issues.

Probably the most important aspect to be considered is the ease of use of the solution employed. The final goal is to cross the finish line faster. Any ramp-up, mis-implementation or debug necessary due to the approach being too complex deepens the issue. With that in mind, half of the effort is to prepare the TB architecture to allow external control and to lower the complexity of the environments.

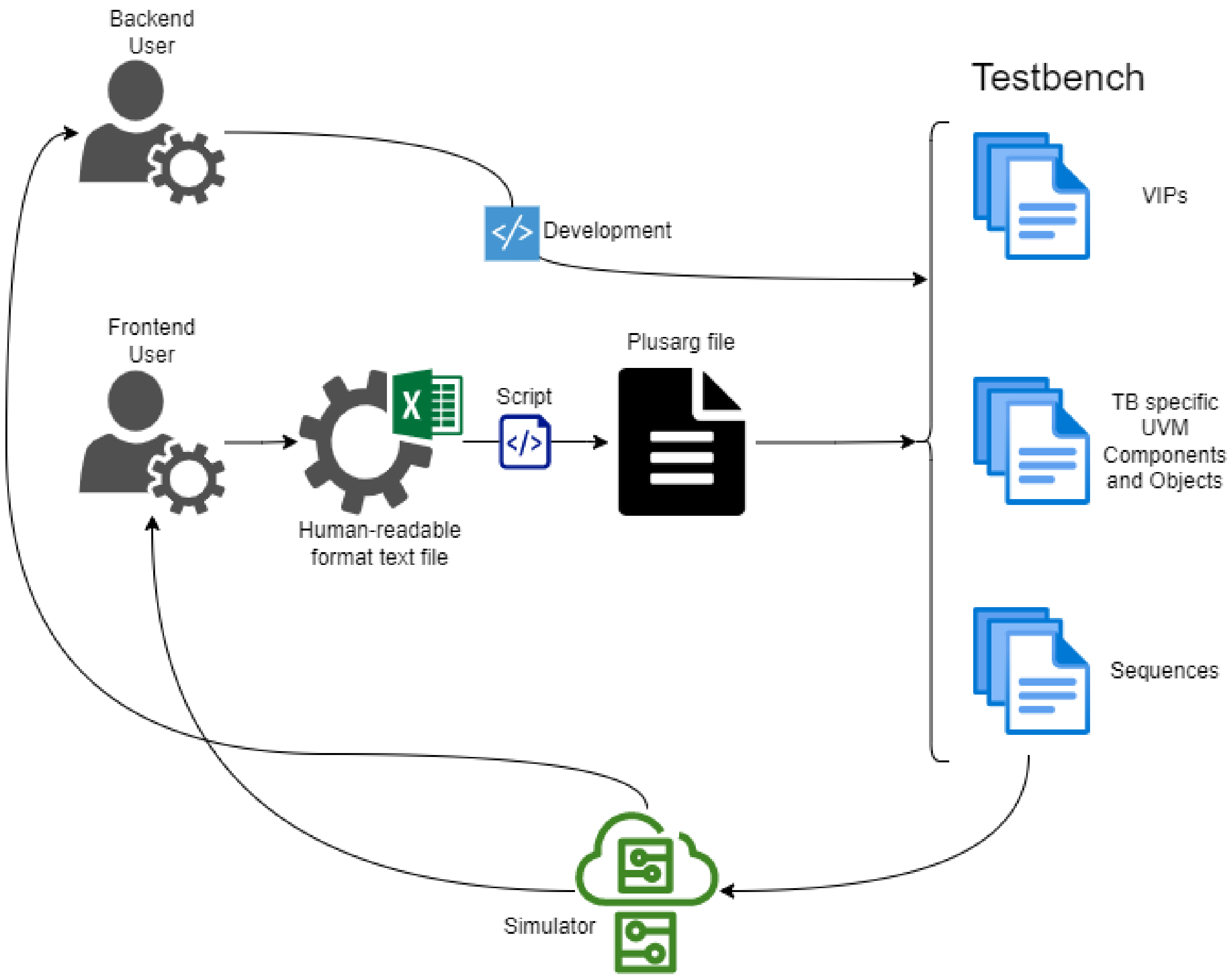

Figure 4 TB control abstraction. Separation of functionality development vs test development

The requirements we came with are the following and the implementation we have is documented in [2]:

- The architecture must be variable and must allow the insertion and removal of components dynamically, easily controllable by an outside script
- All objects and components should have their fields controllable at runtime by an outside script (Example: configuration objects and control flags)
- All sequences should exist stand-alone and not have a hardcoded instance. All sequences should be scheduled by outside input and be controllable at runtime
- All sequences, same as objects and components, should have their respective fields controllable at runtime by an outside script (Example: control knobs and randomization constraints like interval and distribution weight)
- All these should be available to the user seamlessly with as little effort as possible apart from understanding the way of working

There are multiple ways of providing information to a TB at runtime: reading from a file, plusargs, DPI, etc. In this paper, the one we are going to focus on is the plusarg approach. A series of plusargs is defined and confined in one or more text files. These plusargs control the nature of the environment, all object instances and their fields as well as the scheduling and variables of the sequences composing the testcase.

Many times, the way a TB is treated is as a live element that is continuously changing. Different sequences, objects and components are being constantly modified to adjust behavior of certain testcases. This constant change goes against the mindset of software development, mindset that should be applied to testbenches.

The proper way to think of a TB is like an API, a list of functionalities that can be unlocked through different configuration and sequence randomization and scheduling. With that in mind, we can say that the development of a TB is like backend development of software. As a counterpart, testcase creation, debugging, and coverage closure is the frontend. These two should not be mixed. You either implement functionality, or you create/modify test cases, or stimuli. (Figure 4)

The approach we propose is to pull out the frontend of verification outside the scope of the TB. That way we can have a tighter control over the functionality and cleanness of the implementation. Every bit of functionality is much more stable and bug free if it is isolated. The more complexity one adds to an implementation, the harder it will be to maintain. That's why the architecture we propose steers away from hardcoded hierarchical connections tangled functionality. Every agent, every configuration object or component as well as every virtual sequence should exist standalone and implement their own functionality. They should be controlled by setting their respective fields to desired values without expectations of additional dependencies. That's how a TB can have a truly abstract control layer and how it can reach the level of stability that a software implementation has when controlled through an API.

V. AUTOMATED STIMULI ADJUSTMENT

Building on top of the ideas in the previous chapter, the frontend development cycle can be easily and conveniently automated. We already know that the input to the environment can be automatized because we've already proposed the inputs to be organized in a spreadsheet and translated by a script into plusargs. Everything else we need, is a way to parse and translate the results of testcases into data that can be used to generate new/updated inputs. The principle behind this mechanism is depicted in Figure 5.

The existence of such a process opens the way for automatic closure of verification projects by shifting the work from manual iterations to smart feedback loop systems. The engineer should define rules by which the outputs of a simulation/regression should influence future iterations. Based on the nature of the DUT, this can be hard or easy, but in any case, it will take away some of the grunt work of a verification effort. The remaining work would be debugging, something that we can never get truly rid of. However, even if it is not removed, the isolation of functionality is the best approach to reduce the TB debugging effort. Implementing a complex system can be hard and time consuming, but if you break it into atomic parts and implement those, it can become much easier. That is in fact, part of the beauty of the idea of sequencing which is so spread in verification.

One of the building blocks of constraint random verification is in fact the sequence. In the past, people have tried to optimize the constraints applied on sequences during a simulation. By analyzing the results of already-run transactions, the constraints can be adapted to lower the exploration of already covered space. The problem of parsing the results during a simulation is that the computing effort is the same as doing it for an entire regression. However, the impact of parsing results after a couple of transactions is nowhere near as beneficial as doing it after a regression.

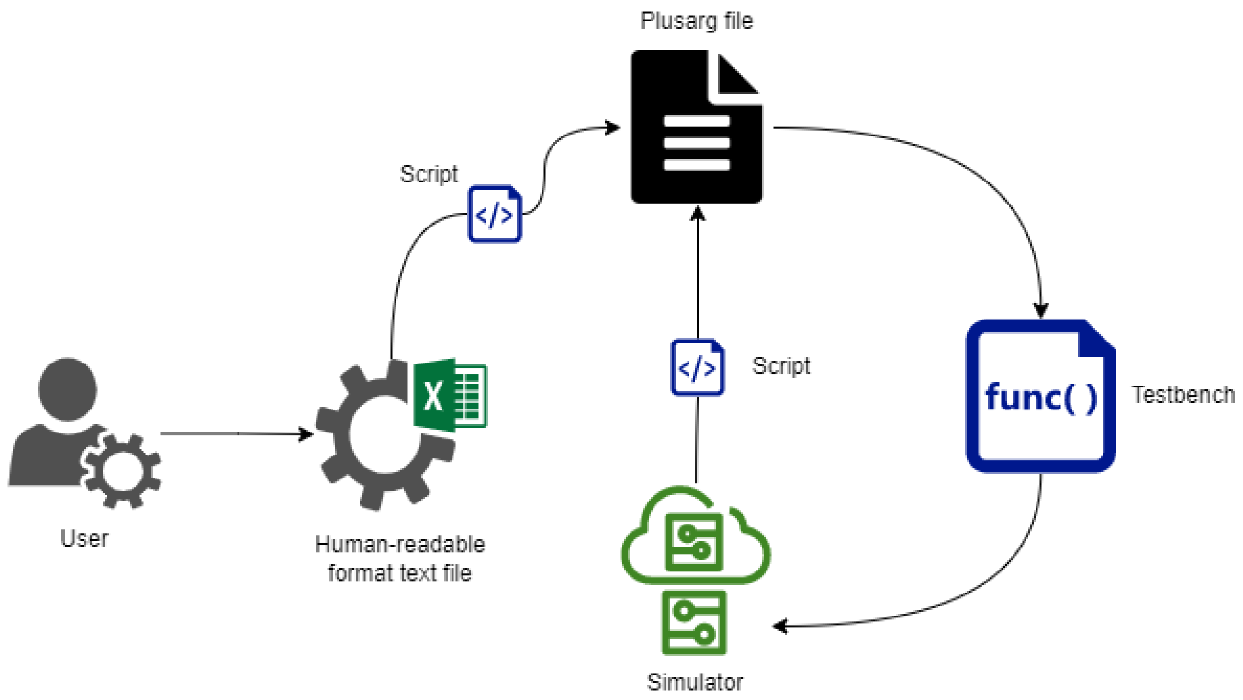

Figure 5. Automatic input adjustments

The proper way to create a feedback loop for a verification project would be to do the adjustments after executing a testcase or even after completing an entire regression. This is because the length of a testcase or a regression can always be adjusted, and you want to allow each constraint collection to do a proper exploration of its space before changing it again.

The road to future enhancement is also vast. One can always have the script that parses the results, create directed testcases for holes, or isolate failing stimulus. The debugging process is also shifted, because now you have a list of abstract variables that describe the contents of the failing testcase. The improvements over a standard methodology are limited only by imagination … or the scripting skills of the user.

VI. Coverage driven Constraint Adjustments / Algorithm Example

As depicted in Figure 5, a feedback loop can be established between the results of a simulation/regression and the plusargs that control the behavior of a testbench. The development cycle as well as the closure stages of a verification project consists of iterations in which the checkers, coverage and stimuli is adjusted. There should be an emphasis on the functionality of the testbench early in the project. Functionality should be implemented as an API that can be accessed via control knobs.

This way-of-working promotes an easier and possibly automated way of controlling this API without having to modify the functionality of the testbench. The control knobs that can be modified through plusargs will allow the TB to change its nature without recompilation and in response to things like pass-rate and coverage. Depicted in Figure 6 is an automated iterative process of adjusting the stimuli based on the merged coverage database of previously run regressions. It is implied that a mature TB should be available that provides all the necessary variables to explore the verification space as per the needs of the project.

The purpose of such a system is for the engineer working on project closure to have a supervisor role over the activity of the TB rather than having to do continuous manual adjustments.

With all these in mind, the steps of a project from development to closure would be the following:

- Define the necessary features and default constraint boundaries that the TB should have
- Implement all the features as an API, where each feature of the TB can be enabled or disabled via control knobs

- Implement completely independent virtual sequences that apply constraints with variable intervals and weights registered as plusargs
- Define a set of initial or general values for all the plusargs (This can be organized in an excel sheet and automatically translated to plusargs via a script as in Figure 5)
- Implement a script that parses UCIS databases Ref [4] and automatically updates the plusargs between every regression run

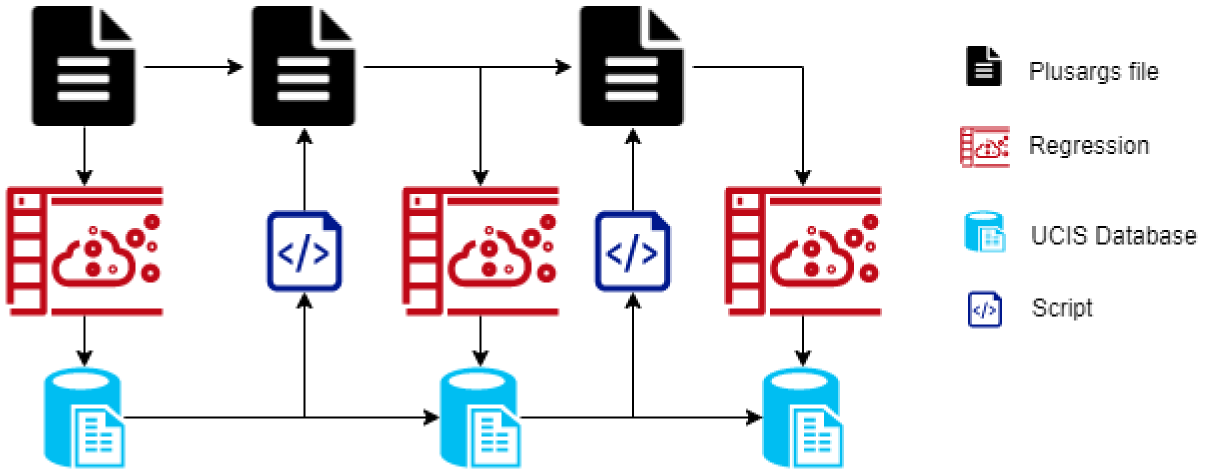

Figure 6. Recursive constraint adjustment based on regression results

When it comes to translating coverage results to plusargs, any number of algorithms can be employed. There is no universal approach to improving stimuli for coverage closure. While there have been several attempts at deploying artificial intelligence algorithms, they are not general enough to be feasible.

In our examples we went with a simple algorithm to prove the concept works. We made all the weights and interval limits for our constraints controllable through plusargs. We chose a number of three integers that have a total exploration space of 9.9035203e+27 values. We then sampled all the randomized values in large cover-points consisting of many bins (1000 per integer). Furthermore, we created a cross bin out of the three (1 million bins). This is made to emulate a constraint-vs-coverage problem that goes beyond what you would find in a regular project.

From the get-go, the coverage goes up to 20% running a first regression. The problem is that the increase in coverage past the first regression is very slow without modifying the constraints. The simplest way to make an automatic optimization is to decrease the chance of the bins that were already covered to be hit again.

In the examples available in the presentation, the weights are dynamically adjusted between every single regression to reach hard to hit bins. This is a completely automatic process, and it doesn't narrow down the randomization space. The purpose of constraint random is to always allow the stimuli to go beyond the "interesting" values that are to be covered. With this approach, certain intervals which have not been exercised yet are given a much higher chance of being randomized. Furthermore, this is an iterative process that evolves with each new regression run, so the changes are not permanent and they don't require any input from the verifier.

VII. Conclusions and Future Research Directions

The best way to shorten project's closure time is by lowering complexity. This has always been one of the most important subjects of discussion in software development: how can one improve functionality while keeping the complexity steady? As our research shows, isolation is a strategy that enables a more efficient implementation of scalable and maintainable VEs. The better one can divide the problem, and the smaller the implementation chunks get, the easier it is to maintain and improve. The Greeks said it over two millennia ago and the words still stand: *divide and conquer*.

For this approach to be successful in verification, you need to always build up, but too many TB hierarchical layers make it too complex. To alleviate the impact of this problem, past a certain level of abstraction, the implementation must be moved out of the target language (SV) and be exposed to other programming languages and tools. This also means that the TB implementation can be isolated from the iterative processes of verification. This offers a certain degree of protection and a tighter control over what goes in.

All the examples that we have already done and some that we are still preparing are meant to prove that there is still a way to improve the verification as we know it today. The focus is on simple software concepts that can be applied to UVM-SV without any licensed tools.

We do have on-going internal research for automatically optimizing coverage closure and we are going to present an example while steering away from any vendor specific tools. We intend to address only topics that are part of open-source standards and tools that we can provide as deliverables. The entire methodology IP that we developed, as well as example use cases, together with scripts that establish the automatization principles discussed in this paper will be provided as an open-source git repository. [5]

REFERENCES

[1] 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language

[2] Andrei Vintila, Sergiu Duda, "How creativity kills reuse – A modern take on UVM-SV TB architectures" DVCon EU 2022 paper. (Proceedings will be available online December 2021, Not published yet at the time of submitting this abstract)

[3] 1800.2-2017 IEEE Standard for Universal Verification Methodology Language Reference Manual

[4] Unified Coverage Interoperability Standard (UCIS), Accellera Working Group

[5] https://github.com/amiq-consulting/amiq_ectb.git