Accelerating Device Sign-off through a Unified Environment for Design Verification, Silicon Validation, and ATE with PSS

Maximilian Suckert
Advantest Europe GmbH
Germany

Sergey Khaikin
Cadence Design Systems, Inc.
United States

Arjun Ashok Vazhayil
Qualcomm Technologies, Inc.
United States

Nandeep Devendra
Qualcomm Technologies, Inc.
United States

Abhijeet Samudra
Advantest America, Inc.
Germany

Klaus-Dieter Hilliges
Advantest Europe GmbH
Germany

Moshik Rubin
Cadence Design Systems, Inc.
Israel

***Abstract*- Solving digital test challenges during device sign-off due to increasing complexity of SoCs, requires the collaboration of design verification, silicon validation, and test engineering teams. In this paper we present a unified hardware and software environment, based on the Portable Test and Stimulus Standard (PSS), for the development and post silicon execution of parameterized functional digital tests. This unified platform approach fosters collaboration between the engineering teams, translating to better Time-to-Quality (TTQ) while reducing Time-to-Market (TTM).**

I. INTRODUCTION

Increasing design complexity, driven by advanced technology nodes and new packaging technologies for digital devices, requires new and innovative testing approaches to overcome the time-to-market pressure of today. These new demands are observed across all value chain steps and push pre-silicon Design Verification (DV), Silicon Validation (SV), and test engineering (TE) teams to work together during device sign-off. The lack of a systematic flow from pre-silicon design verification to silicon validation in either a lab bench environment, or an Automated Test Equipment (ATE) environment, creates inconsistency, inefficiency, and correlation issues. To bridge this gap, we propose a unified environment for device bring-up and operation during test. This environment, based on the Portable Test and Stimulus standard (PSS), enables DV teams to both directly execute tests on silicon as well as enable collaboration with SV and TE teams by sharing test content.

A. Improvements Introduced by the Unified, PSS Based HW and SW Environment

The unified environment (Figure 1a) enables DV to complement pre-silicon efforts in a post silicon bench environment to increase functional coverage and to accelerate coverage closure. DV benefits from faster execution on silicon than on simulators and emulators. Sign-off and debugging of test content that cannot be simulated correctly, such as threading behavior or precise power consumption, can conveniently be done on silicon.

SV engineering teams benefit similarly from pre-silicon coverage concepts that are introduced by the PSS to evaluate runtime coverage for execution on silicon. The unified environment enables DV and SV teams to jointly bring up and debug IP on silicon. A first-time-right for tests is enabled by PSS verification IP, which can be verified in simulation before sharing the tests for silicon validation.

After completion of device sign-off, DV can provide validated targeted functional tests to TE for high volume manufacturing (HVM) on ATE. These tests come with well-known defect coverage derived from the PSS model of the test intent. The details of the flow to TE within the unified environment are described in [1].

B. Challenges and Solutions for Bringing up a Unified Environment

Today, DV, SV, and TE teams use different tools and formats for test content. Their workflows are separated. PSS modeling introduces an abstraction layer to decouple test intent from test realization, and to describe a test at SoC level. The unified environment introduces well-defined APIs for controlling test execution on silicon and for packing test content to enable seamless portability between teams.

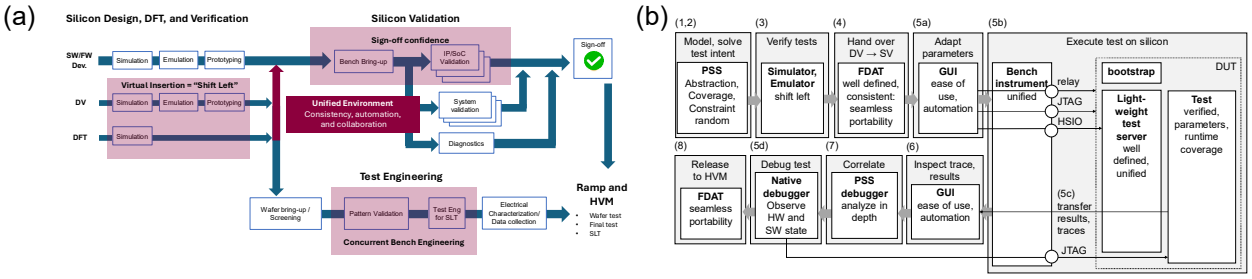

Figure 1. **(a)** Unified software and hardware environment for DV, SV, and TE. **(b)** Components and work steps of the streamlined workflow based on the PSS and on the unified software and hardware tools. Grey boxes indicate the work steps, white boxes the components of the unified environment, and circles the HW interfaces of the bench instrument. Numbers refer to the text.

Complex and non-standardized post-silicon bench setups operated by SV are often comprised of instruments from multiple vendors. These setups cannot be used by DV teams for reliable and repeatable execution of tests without support from SV experts. The unified environment introduces a bench instrument with industry-standard interfaces and software for instrument configuration and test execution. The software environment features a GUI and a scripting API. After initial setup of the bench by an SV engineer, DV teams can execute and debug tests independently on silicon.

ATEs, which feature similarly well standardized interfaces, support reliable and repeatable execution, but are both complex and costly. Industrial power and cooling requirements are physical limitations which prevent scaling in SV and TE laboratories. Often production equipment must be shared, resulting in ATE test time as a scarce resource. ATE software is designed to optimize test times and to efficiently use ATE instruments, requiring support from TE experts for operation. In contrast, the unified bench instrument with air-cooling, standard power supply, a bench footprint, and a software API, which is tailored for device and test sign-off, allows for scaling to multiple consistent setups even in facilities with no industrial capabilities.

For data collection during device characterization, and for HVM tests, test binaries must be converted to ATE vector patterns. This conversion requires long-running simulations, for each adaptation of the test binaries. The execution of patterns suffers from poor observability of HW and SW behavior. Correlating pattern failures with actual HW and test SW is tedious and involves debugging sessions with many domain experts. The unified PSS based SW environment and the bench instrument enable native in-system debugging, and a correlation of PSS test intent with test traces. In the unified SW environment, we introduce a PSS API for runtime parameters and runtime results. Runtime parameters and results allow exploring the test parameter space with a systematic, coverage driven, and automated approach.

II. Seamless, Well Specified, Unified Hardware and Software Flow

In this section, we introduce in detail the unified environment that enables DV, SV, and TE engineers to efficiently collaborate during device sign-off. Efficient collaboration is achieved by using common methodologies for creating, sharing, and debugging test content, and for analyzing test results and coverage.

Figure 1b shows the steps of the workflow and the components of the unified environment, which support the workflow. The unified environment supports the following steps of a unified workflow across DV, SV, and TE. (1) The DV team models test intent and coverage with PSS. The test intent includes bring-up and functional test of SoC IP. A dedicated PSS API layer supports modeling of test intent with runtime parameters and results. (2) The PSS tool derives solutions within the specified constraint and coverage statements, and generates target code. (3) The DV team verifies the test (solution for test intent) and the silicon execution environment (for example boot, test loading) in a simulation and emulation environment. (4) The DV team transfers the test to the SV bench using a functional test data (FDAT) container that supports seamless portability. Next, DV and SV teams validate the test on the SV bench. This includes (5a) exploring the behavior of test across the test parameter space, (5b) executing the test, (5c) recording traces and capturing results, and (5d) debugging the test and inspecting the state of the DUT using a native SW debugger. A unified bench software with a GUI and scripting interface supports these steps. (6) DV and SV engineers visualize and analyze traces and results within the GUI. (7) DV and SV engineers correlate traces and tests in the PSS tool. (8) After sign-off, tests are shared with TE for HVM using the FDAT as outlined in [1].

A. Native PSS API Layer for Runtime Parameters and Results and PSS Eco System

Modeling test intent with PSS and separating test intent from test implementation for specific targets, introduces an abstraction level that eases portability of tests and methodologies between different teams (DV, SV, TE). The

teams might work with different targets (simulation, emulation, prototyping, post silicon) and with different projects.

PSS enables users to abstract and solve test intent within given constraints and coverage targets. A PSS solver provides a set of valid and consistent tests for a given test intent, meeting the coverage targets. The EDA (Electronic Design Automation) ecosystems for PSS streamline test analysis with debuggers for the solve flow, and with graphical tools for visualizing tests and correlating runtime traces with tests. PSS enables measuring coverage for functional tests on post silicon and introduces a unified interface for tracing the test behavior.

In this work, we implemented a PSS API layer for modeling of runtime test parameters and runtime test results, using native PSS language features. Runtime test parameters allow users to modify the behavior of a test at runtime. Capturing runtime test results (for example sensor readouts, register status) as a function of test parameters enables exploring test runtime behavior. Users can readily analyze the captured structured results without having to extract, format, or clean the data from test traces or from iterative register dumps.

As a first step before modeling runtime parameters and results, test modelers select the parameters and results of the test that should be made available at runtime for modification and capture. Resembling the architecture of SoCs in components, the API organizes these parameters and results in sets per component (Figure 2). Users declare the sets as PSS structs, derived from a dedicated base struct type, in the user's PSS components (Figure 2a). Ranges for the set members are given with PSS constraints. For publishing descriptions of the set members, we use a PSS tool-specific feature (*@tooltip* of *Cadence Perspec System Verifier*). To initialize the parameters and to trigger target code generation, users schedule an initialization action for all struct instances (Figure 2b). Users reference the sets in their test implementations (PSS action *exec* blocks) by value or by handle. Values and handles are provided from getter functions. To link getter functions and PSS structs of a set, we adopt the design pattern of the PSS core library for registers, referring the getter and the PSS struct within a dedicated PSS component. With this approach, users can model sets following a familiar concept. Forwarding handles to sets for complete components has proven beneficial for the interface between PSS test writers and implementers of C target functions. This is because adding a parameter of a target function does not require to adapt function signatures in PSS code, C sources, and C headers (Figure 2e).

The PSS API generates C code that describes the memory layout of the runtime parameters and results (Figure 2c, d). For each set, we generate getter functions that provide pointers to the memory location of the set. For target code generation, we inspect in a first step the PSS model of the sets using the reflection API specifically available in *Cadence Perspec Verifier*. To this end, we inspect in the *post_solve* step the properties of all instances of the structs that are derived from the base type for runtime parameters and results. In a second step, we generate the target code,

(a) PSS

```
package example_pkg {
  // packages for runtime parameter and result sets
  import psv_vip_pkg::*;
  import psv_vip_target_types_pkg::*;

  // declare parameter set
  struct user_param_set_algorithm_s : psv_base_param_set_s {
    //@tooltip("Algorithm type: Fast = 0, Slow = 1")          // description
    rand uint8_t algorithm_p;                                 // name and type
    constraint algorithm_p_min_max { algorithm_p in [0..1]; } // value range

    //@tooltip("Seed")
    rand uint16_t seed_p;
    constraint seed_p_min_max { seed_p in [0..UINT16_MAX]; }
  }
  // declare parameterized component
  component user_algorithm_c {
    // define parameter-set component
    psv_param_set_c<user_param_set_algorithm_s> user_param_set_c;
    action initialize_a {
      activity {
        // initialize parameter set user_algorithm::user_param_set_c
        do psv_param_set_c<user_param_set_algorithm_s>::psv_initialize_param_set_a with {
          comp == this.comp.user_param_set_c;
          // default value
          param_set.algorithm_p == 0;
          // default value of param_set.seed_p can be selected by solver
        };
      }
    }
    // user target functions with test implementation
    target function void runAlgorithm(chandle param_set_pointer);
    import target C function runAlgorithm;
    action run_test_pointer_a {
      exec body {
        // get pointer to parameter set
        comp.runAlgorithm( comp.user_param_set_c.get_pointer() );
      }
    }
  }
}
```

(b) PSS

```
package test_pkg {
  // packages for runtime parameter and result sets
  import psv_vip_pkg::*;
  import psv_vip_target_types_pkg::*;
  import example_pkg::*;
  extend component pss_top {
    // instantiate parameterized components
    user_algorithm_c algorithm1;
    user_algorithm_c algorithm2;
    action main_a {
      activity {
        // initialize parameter sets of pss_top::algorithm1/2
        do user_algorithm_c::initialize_a with { comp == this.comp.algorithm1 };
        do user_algorithm_c::initialize_a with { comp == this.comp.algorithm2 };
        // generate target code for parameter sets
        do psv_vip_parameters_c::prepare_parameters_a;
        // run test
        do user_algorithm_c::run_test_pointer_a with { comp == this.comp.algorithm1; };
        do user_algorithm_c::run_test_pointer_a with { comp == this.comp.algorithm2; };
      }
    }
  }
}
```

(c) Generated C code

```
// C code describing parameter layout in DUT memory
// generated by PSS API for parameter sets
struct TesterParameters {
  uint16_t pss_top__algorithm1__user_param_set_c__algorithm_p; /*[Min: 0; Max: 1; Help: Algor .. Slow = 1]*/
  uint16_t pss_top__algorithm1__user_param_set_c__seed_p; /*[Min: 0; Max: 65535; Help: Seed]*/
  uint16_t pss_top__algorithm2__user_param_set_c__algorithm_p; /*[Min: 0; Max: 1; Help: Algor .. Slow = 1]*/
  uint16_t pss_top__algorithm2__user_param_set_c__seed_p; /*[Min: 0; Max: 65535; Help: Seed]*/
};
```

(d) Generated C code

```
// declaration of structs of parameter sets
// generated by PSS API for parameter sets
typedef struct dut_algorithm_pkg_user_param_algorithm_struct dut_algorithm_pkg_user_param_algorithm_s;
struct dut_algorithm_pkg_user_param_algorithm_struct {
  unsigned short algorithm_p;
  unsigned short seed_p;
};
```

(e) User C code

```
// target function using a pointer to a parameter set
// user code
void runAlgorithm ( void * pointer ) {
  // cast pointer to struct
 dut_algorithm_pkg_user_param_algorithm_s* parameters = (dut_algorithm_pkg_user_param_algorithm_s*) pointer;
  // test implementation
  printf("seed: %u", parameters->seed_p)
}
```

Figure 2. **(a, b)** Example of modeling a test with a component with a runtime parameter set with PSS API layer. **(c)** Example of generated target C code, describing memory layout of parameter in DUT memory and **(d)** declaring struct types for parameter sets. **(e)** Example of user code using a pointer to a parameter set.

from the information retrieved with reflection, in dedicated C files with the file-IO API of PSS. To ease correlating set members in the generated code with the PSS model, we generate the set members with names that allow to identify the instances of the components with the sets. To this end, we include in the name of a set member the names of the component instances in the component tree (from the component with the set up to the root component).

Using the PSS runtime API on top of PSS leverages all benefits of the PSS eco system. Since native constructs of PSS are used, the API supports fast learning and straightforward integration into existing models. Using reflection as the base of code generation ensures that consistent target code is produced without any further user interaction beyond PSS modeling.

B. Well-defined Binary Interface for Runtime Parameters and Results

For reliable modification of runtime parameters and capture of runtime results, we defined a simple binary interface. The new bench SW environment supports dedicated memory sections, which hold the parameters and results. The memory addresses of the sections are defined during the build of the test sources, in the linking phase, and stored in the ELF binaries. Users declare the memory sections in their linker scripts or scatter files.

The bench SW environment transfers parameter and result data between the bench instrument and the DUT memory according to the memory addresses defined in the test binaries. Dedicated scripting for the transfer of parameters and results is not required.

C. Seamless Portability of Test Content with FDAT

For seamless, self-contained portability of test content between DV, SV, and TE engineering teams, we introduce the functional test data container (FDAT). The FDAT is a well-defined, test insertion agnostic interface for transfer of tests [2]. An FDAT holds the test binaries, a scheduling sequence for these, and optionally, source files for debugging. The FDAT is extendable with user content such as documentation, test version information, or trace decoding information. Thus, the FDAT supports users with adding flow enhancements and transferring supplementary artifacts. This extensibility enables integration of the unified environment with existing user test frameworks and tools.

The FDAT is a zip container with a one-level directory structure. Users can quickly set up the packaging of test content in their existing, typically scripted test generation frameworks. The scheduling sequence is declared with XML and can be generated conveniently for most use cases from templates.

D. GUI and Scripting Frontend for Automated, Systematic Exploration of Runtime Parameter Space

We introduce a software framework for test handling and automated execution of tests with the new bench instrument. The framework supports organization of tests in projects, configuration of the bench instrument, and automated, systematic exploration of the parameter space with batch and Shmoo test execution. A GUI frontend enables fast turnaround, interactive exploration, and debugging tasks. With the scripting API, users can automate routine tasks and can integrate project handling and the bench instrument in their test frameworks. The backend can control a bench instrument and an ATE to collect data for comparing test behavior on bench and ATE.

E. A Unified, Reliable Bench instrument

A unified and reliable bench instrument is key for scaling the flow within a project, across projects, and across DV, SV, and TE teams. To enable scaling, we integrate all interfaces that are required for stimulating and monitoring a DUT in a new single bench instrument, which features the reliability and execution speed of an ATE. To fit seamlessly in an SV lab environment, this instrument features industry standard connectors for the interfaces, a bench footprint, and air cooling.

F. Industry Standard Hardware Interfaces

For conveniently starting and controlling the DUT and the test server (see II-G), for downloading tests, and for observing the software and hardware during the test, the new bench instrument features industry-standard debug and control interfaces, and native HSIO interfaces of the DUT.

We use a debug interface (such as JTAG, SPI, UART, or I2C) for bootstrapping the DUT, downloading HSIO drivers and the test server, and for attaching a software debugger. We use an HSIO interface for downloading tests, controlling the test server, and transferring traces and results of the test activity. Support for native HSIO interfaces (USB, PCIe, ETH) enables a direct transfer of test binaries to the DUT memory. In contrast, high pin-count parallel test interfaces stimulated with ATE digital cards require a cyclization of binaries into ATE patterns.

G. Well-defined Interfaces for Reliable Handling of Tests on DUT

Well-defined interfaces for transfer of tests, parameters, traces, and results ensure a consistent first-time-right setup of tests, reliable observability of test results, and capture of traces. We developed a lightweight test server, which interfaces, on the DUT, the communication on debug and HSIO interfaces between the bench instrument and the test [3].

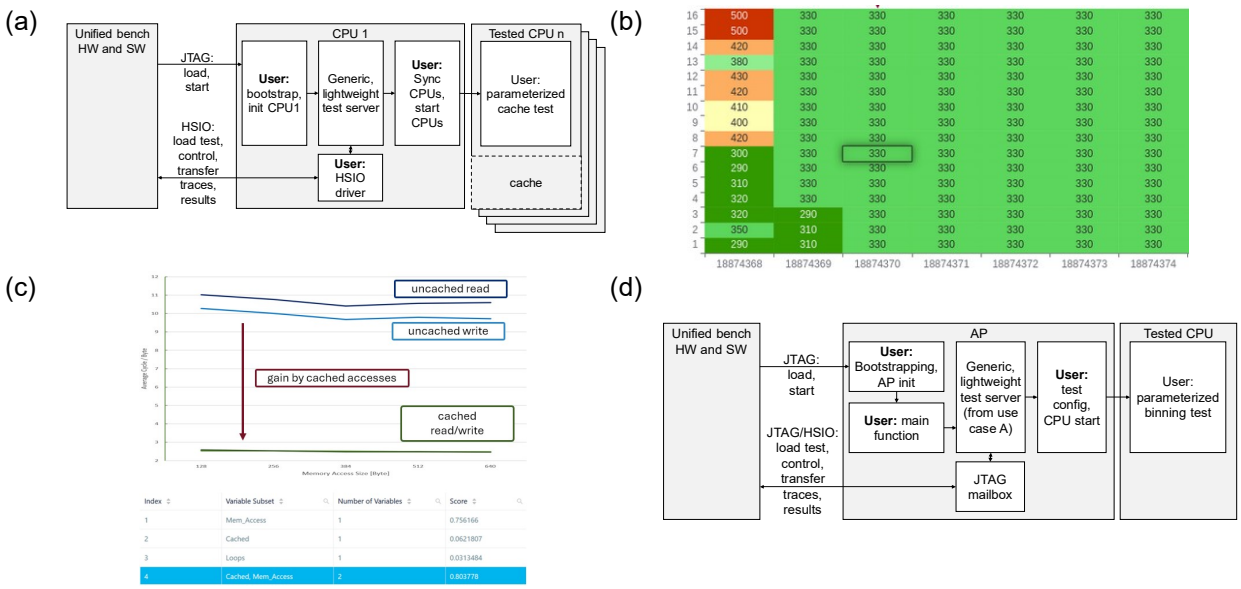

Figure 3. **(a)** Block diagram of use case A. Grey boxes refer to hardware components, white boxes to software components. **(b)** Shmoo plot display. **(c)** Most influential parameter set ranking. **(d)** Block diagram of use case B. (AP is an always-on processor.)

For communication between bench SW and the test server, we introduce a simple high-speed protocol that supports data transfer to and from DUT memory, runtime messaging, and control of test execution [4]. The test server interfaces with the test for messaging and execution control via a simple *syscall* interface. This interface comprises functions for sending trace messages and returning an exit code.

For a user device, the (fully modifiable) code for the core functionality of the test server must be integrated with device-specific HSIO drivers. In an early project phase, if HSIO drivers are not yet available, communication via debug interfaces is supported.

H. Observability of Software and Hardware

For exploring and debugging the test software behavior and for identification of hardware issues, comprehensive observability and a clear display of captured results and traces as a function of the test parameters are decisive for efficient analysis and correlation.

The GUI features viewers for visual inspection of activity traces and test results right after execution of a test, a Shmoo, or a batch run. Pre-configured plotting views support inspection of margins or easy identification of test failures and outliers for follow-up analysis in an EDA scenario viewer. Similarly, analysis tools, such as a ranking of most influential parameter sets of a test, can provide indications of HW issues. EDA scenario viewers feature correlation of activity traces with test scenarios and a graphical display of the trace. From the analysis of a trace in the EDA tool, engineers can pinpoint where to set breakpoints during in-system debugging.

Runtime messaging enables collection of data for runtime coverage analysis with EDA tools. Because of faster execution on silicon in contrast to simulation, coverage closure can be accelerated.

I. Native SW Debugging Environment

A native SW debugger enables engineers to step through test code, to inspect the device state, and to thereby analyze misbehavior of test and DUT HW. The integrated support for industry standard software debuggers, such as Lauterbach TRACE32, in the test SW environment and the bench instrument enables DV engineers to debug their tests without support from an SV engineer.

III. RESULTS

A. Use Case: Parameter Space Exploration and Data Analysis

For validation of the unified flow, we modeled a complex multicore cache access test scenario in PSS. We integrated the test server with drivers for a Xilinx Zynq-UltraScale+MPSoC—configured with 4 Cortex A53 cores, MMU, DDR, and SRAM—to communicate via JTAG and USB (Figure 3a). We used the runtime PSS API layer to seamlessly parameterize verification IP of Cadence for ARM core testing and to capture results.

With the GUI frontend, we scheduled a Shmoo for exploring the behavior of the test across the parameter space (Figure 3b). In the GUI, we ran an analysis on the collected data to identify the set of most influential parameters with a ranking according to a score that reflects the influence of the parameter sets on the collected results (Figure 3c).

B. *Customer Use Case: Configuration and Observability of a CPU Binning Test*

For a customer use case, DV and SV engineering teams leveraged the unified HW and SW environment to collaborate seamlessly to prepare, bring up, and debug CPU binning of an SoC device (Figure 3d). Optimization of CPU binning requires exploring the behavior of selected CPUs with respect to specific operating parameters. PSS enabled the DV team to favorably abstract and schedule the IP for selecting the tested CPU and for configuring the frequency corner and the memory range used during the test. With the runtime parameter PSS API and the FDAT, the DV team could readily publish the configuration parameters, including documentation, to the SV engineers. The result parameter API and PSS messaging enabled capturing the pass/fail status of the test and collecting data for confidently signing off the test, and for identifying critical corners of the parameter space. Signing off tests with a bench instrument rather than with an ATE removed the need for cyclization of binaries. Tracing and connecting a debugger accelerated validation of the test behavior and identification of programming issues during the bring-up of the binning test.

The test server was executed on an always-on processor (AP). A first test binary was transferred to the DUT and ran on the AP to start the CPU selected for the test. This first test starts a second test on the selected CPU to determine its performance as a function of the operating parameters. For implementing the test schedule, the team reused the example code of the test server core functionality used for *use case A* and built the code for the target SoC instruction set. The DV team adapted the interfaces of the SoC specific embedded software components to the specification of the test server (see section II-G). This included adapting linker scripts, linking existing bootstrapping code and AP initialization to the test server, adapting the main function of the test, and attaching PSS-templated callbacks for messaging to the messaging interface of the test server.

Because no HSIO driver was available in the early validation phase, communication with the bench instrument was done using the JTAG debug interface. In the next step of the validation phase, a seamless switch to the SoC's USB interface could be done once the HSIO drivers were available.

Scheduling of the loading of the binaries and initial conditioning of the DUT (not yet done in the early project phase by executing code on the AP, but rather with parameterized Lauterbach PRACTICE scripts) was implemented in an FDAT sequence, which the DV team generated from templates. This pragmatic approach, reusing existing scripts, sped up the bring-up and enhanced acceptance of the flow in the DV team.

Introducing PSS and the FDAT interface simplified and streamlined the handover of the tests from DV to SV. The parameter interface abstracts the test behavior from register-level programming and allows capturing behavior as a function of test parameters, rather than delivering a dedicated test per parameter set. The FDAT includes documentation of the parameters, so that an error prone handover of parameter lists is avoided. The runtime PSS API is written in pure PSS, so that integration onto the customer's existing PSS methodology-based verification environment was straightforward. In combination with the FDAT container, this allows making verification test content easily available for execution on silicon. The initial effort for introducing the FDAT container, adapting linker scripts, and integrating the test server was eased by learning from examples and by using templates.

During bring-up of the test, the teams benefitted from streamlined feedback from SV to DV and from the enhanced observability of SW and HW (registers) behavior. This was achieved using the runtime capture of test traces, in contrast to register dumps at specific break points.

Attaching a Lauterbach TRACE32 debugger to the DUT from the GUI—using the readily available debugging dongle in the bench instrument, configured once by the SV team at the project start—enabled the DV team to debug the test independently from lab hardware experts from the SV team. The unified flow allowed validating the test on the SV bench. This removed the need for digital pattern debug with poor observability on ATE of software and hardware issues.

The abstraction of IP, runtime parameters, runtime results, and trace capturing, which enables observability and correlation of traces and test intent, are all modeled purely in PSS and transferred in an FDAT. This method can be reused and scaled for future SoC projects, unlike the less abstract and poorly scaling C code used previously.

C. *Accelerating Device and Test Sign-off*

Applying the unified hardware and software environment for the development and post silicon execution of parameterized functional tests eased collaboration between DV and SV engineering teams in the use case presented. From this use case, we can derive estimates for the acceleration of device and test sign-off.

For large PSS test models with many constraints, a solve run may take hours per solution. Leveraging run-time parameters and automated execution allows DV teams to explore the parameter space and to collect coverage on silicon faster than at solve time. Execution on silicon is faster by a factor 10^5 to 10^6 compared to simulation, so that coverage closure can be accelerated. In addition, execution on silicon allows executing tests that cannot be simulated with sufficient accuracy, such as threading behavior or precise power consumption, so that runtime coverage can enhance overall coverage.

The unified environment with functional interfaces to the DUT enables DV and SV teams to execute test binaries on silicon without converting them to ATE test patterns, and to observe and debug the test behavior using tracing and native software debuggers. The direct execution of binaries avoids hours of RTL simulation during binary-to-pattern conversion required for each debug iteration. The direct observability of the same test content used by DV and SV teams allows identifying SW and HW issues directly from traces or with in-system debuggers. Previously, a joint effort of multiple domain-experts was needed to derive the actual issue from a pattern failure. Avoiding dead times, during handover and when organizing debug meetings across teams, saves hours up to days per debug cycle.

IV. Tools Enabling the Unified Environment

Cadence Perspec System Verifier supports modeling of test intent with PSS 3.0 [5]. PSS introduces abstract modeling and portability of test intent between multiple projects and different target platforms in DV, SV, and TE environments. From the test intent, *Perspec's* PSS solver generates tests, which are correct-by-construction, with a coverage measurement according to the coverage statements defined in the model of the test intent. *Perspec* features tools for debugging of the solve process, and for graphical inspection of test intent and test solutions. The solver of *Perspec* supports reflection of the test solution, which we use to extend PSS constructs with additional generation of target code for the new API layer for runtime parameter and result sets.

SiConic Link is a new bench instrument supporting functional communication for fast and reliable test execution, test activity tracing, and native software debugging. *SiConic Link* introduces the reliability of ATE instruments to SV bench environments, with a small footprint and infrastructure requirements compatible with SV environments. *SiConic Link* features HSIO and debug interfaces for connecting and controlling SoC devices.

SiConic Explorer is a new bench software environment for streamlined execution of parameterized, software driven functional tests with a graphical user interface and a scripting interface. *SiConic Explorer* features configuration of the bench instrument, test management, automated parameter space exploration, graphical inspection of traces, and plotting and analysis of results. *SiConic Explorer* can control both *SiConic Link* and the *Advantest V93000 EXA Scale*.

Cadence Verisium features analysis of activity traces. *Verisium* correlates messages in the activity traces with actions in the solution of the test intent and visualizes activity traces in a waveform-style view.

V. Outlook

The proposed environment features well-defined hardware and software interfaces, allowing users to extend and integrate the unified workflow in existing verification and validation environments. Users can connect the bench instrument to a HW prototyping platform to verify their tests. This approach provides a platform for firmware and software teams for their development, and it enables to prepare tests for a first-time right execution on silicon. Further, users can automate control of the bench software with the scripting interface and integrate the software into their test frameworks. Scripting access to variation of results, test execution, and result collection can enable, for example, implementation of tuning of tests for specific test targets, such as maximum stress.

There is potential for further improvements in accelerating typical engineering tasks in the proposed environment. To accelerate test setup and result analysis tasks, the GUI of the bench software will be extended with additional built-in automation, for example margin search or detection of anomalies in Shmoo analysis, and with more features for data analytics, for example correlation analysis or variance analysis. To streamline monitoring of the DUT state during test debug efforts, we will leverage PSS for sharing SoC descriptions, for example to model registers in PSS as single source.

VI. Conclusion

We demonstrated the implementation of a streamlined flow of pre-silicon content to a post silicon bench environment. Qualcomm teams used this flow successfully to address configuration and optimization challenges for a CPU binning test with quick turnaround times. The initial setup of the new flow was eased by adapting templates and example code, which has been verified on an SoC device in an Advantest lab. The Qualcomm teams benefited from the PSS-based API for parameterization and result capture. This API could be integrated into their modeling infrastructure in a straightforward manner. The FDAT interface, and using functional HW interfaces, allowed them to hand over the test binaries from DV to SV teams directly, without cyclization, for execution on the bench. We estimate that exploring the parameter space on silicon for test optimization, without time consuming iterations of the complete workflow, and native debugging during bring-up of the test, saves the teams hours up to days, per debug cycle, during device sign-off.

The integration of the flow in a PSS-based approach and the unified test environment has proven to break barriers between organizational silos, thereby enhancing productivity. The proposed unified environment defines critical interfaces between the teams and components, and provides APIs for easily connecting the users' PSS modeling and scripting frameworks. This approach introduces reliability while being open to adapt the methodology to the specific needs of DV and SV users. After initial setup of the flow, teams can easily share test content and test results, and

DV engineers are enabled to explore and debug the behavior of the test on silicon independently from SV experts. The fast execution on silicon, in contrast to simulation, helps to reduce the time required for coverage closure. The reduction of execution time by a factor of 10^5 to 10^6, combined with the PSS productivity gain of approximately a factor 5, brought from DV to silicon [6], and the seamless handover to HVM [1], result in significant improvements of the TTM and the TTQ.

ACKNOWLEDGMENT

We thank Marcus Schulze-Westenhorst for support with the analytics use cases and Sarah Rottacker for the automated tuning example. We thank Brian Archer and Pete Hodakievic for good discussions.

REFERENCES

[1] M. Schulze-Westenhorst, S. Jörg, M. Bücker, K.-D. Hilliges, and M. Braun, "A Novel Approach to Functional Test Development and Execution using High-Speed IO," presented at the DVCON Europe, 2021. [Online]. Available: https://dvcon-proceedings.org/document/a-novel-approach-to-functional-test-development-and-execution-using-high-speed-io/

[2] Advantest, "FDAT specification," *Technical Documentation Center*. in press. [Online]. Available: https://www3.advantest.com/service-support/ic-test-systems/products-list/v93000-technical-documentation/

[3] Advantest, "Advantest Test Scheduler," *Technical Documentation Center*. in press. [Online]. Available: https://www3.advantest.com/service-support/ic-test-systems/products-list/v93000-technical-documentation/

[4] Advantest, "HS protocol specification," *Technical Documentation Center*. in press. [Online]. Available: https://www3.advantest.com/service-support/ic-test-systems/products-list/v93000-technical-documentation/

[5] Accellera Systems Initiative, "Portable Test and Stimulus Standard." 2024. [Online]. Available: https://www.accellera.org/downloads/standards/portable-stimulus

[6] S. Vasu, N. Venkatesh, and J. Maitra, "Media Performance Validation in Emulation and Post Silicon Using Portable Stimulus Standard," presented at the DVCON US, 2021. [Online]. Available: https://dvcon-proceedings.org/document/media-performance-validation-in-emulation-and-post-silicon-using-portable-stimulus-standard/