Boost Verification Results by Bridging the Hardware/Software Testbench Gap

Matthew Ballance
Mentor Graphics Corporation
Design Verification Technology Division
Wilsonville, Oregon
matt_ballance@mentor.com

***Abstract*—Today's complex designs increasingly include at least one, and often more, embedded processors. Given software's increasing role in the overall design functionality, it has become increasingly important to leverage the embedded processors in verifying hardware/software interactions during system-level verification. This paper presents a UVM-based package for software-driven verification, and presents applications of this package that enable more-comprehensive system-level verification.**

Keywords—functional verification; software-driven verification; system-level verification; Univeral Verification Methodology; graph-based stimulus

I. INTRODUCTION

Complexity begets complexity. Which is to say, to run properly and fully realize their intended behavior, today's intricate hardware designs with tens of millions (or more) gates inevitably require ever more nuanced and elaborate software. This tight coupling means that the traditional verification process, where both hardware and software are designed and verified in isolation and then integrated late in the design cycle, is less and less workable. What is needed is a two-way street for verifying low-level software and verifying the hardware used by the embedded software, and doing both as early in the process as possible.

A software-driven verification environment differs in some fundamental ways from a hardware-centric verification environment. Figure 1 shows a typical hardware-centric SoC verification environment. Agents within the testbench environment apply stimulus to the design via its interfaces. Software runs on the processor within the design to manage the component IP blocks. From a structural perspective, the processor, and the software running on it, is part of the design.

In the diagram below, the hardware and software portions of the testbench environment run in isolation, which leads to several problems. First, the isolation makes it very difficult or impossible to coordinate the operation of software running on the processor with stimulus provided from the hardware-centric testbench environment. It is also extremely challenging to achieve the level of comprehensive verification needed to ensure proper design operation, since the software-driven stimulus is unable to take advantage of advanced stimulus generation techniques or to contribute to functional coverage metrics.

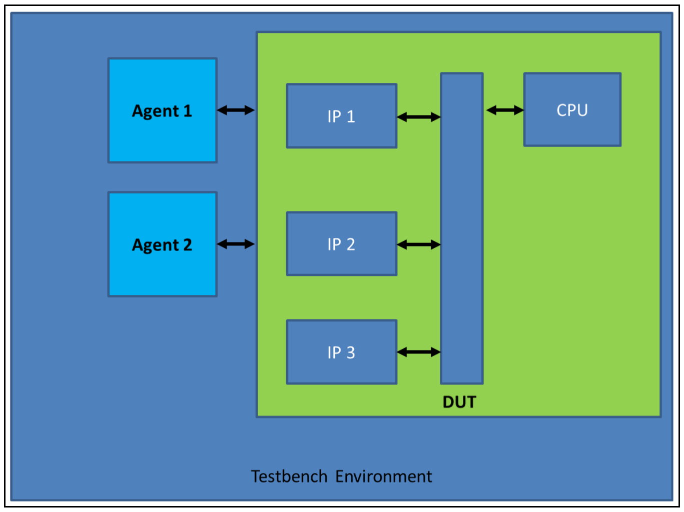

Figure 1 – Hardware-Centric Verification Environment

The comprehensiveness of verification can be increased by making the software running on the processor part of the verification environment. For example, in the case of this design, IP 3 has no external interfaces. Its operation in conjunction with IP 2 can really only be verified by controlling IP 3 via software, while applying stimulus to the external interface of IP 2. Enabling this coordination and cooperation of the hardware-centric and embedded-software portions of the testbench is a key goal of a software-driven verification environment.

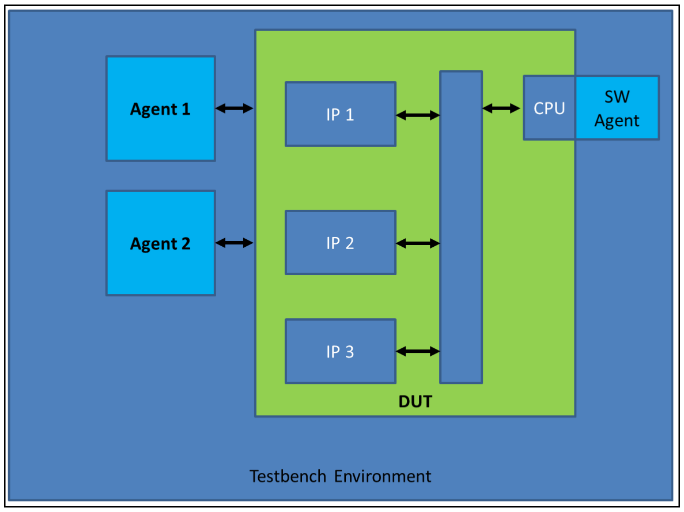

Figure 2 - Software-Driven Verification Environment

Figure 2 shows a block diagram of a software-driven verification environment. Functionally, the embedded software running on the embedded processor becomes another agent in the testbench environment that is able to apply stimulus to the hardware design from the inside.

Some forward-looking verification engineers have created testbench environments with built-in hooks to enable coordination between the hardware and software aspects of the test. Though undeniably useful, these hooks are usually specific to a project or verification team and so typically aren't portable. This paper proposes a methodology and software-driven verification (SDV) extensions to UVM that enable embedded software to participate in hardware-centric verification starting at the block level and continuing to the system level.

II. UVM SDV PACKAGE

The UVM Software-Driven Verification (SDV) package provides infrastructure that enables embedded verification software to coordinate with a UVM testbench environment. The majority of the features provided by the UVM SDV library simply extend features already provided by the UVM library into the software domain.

The UVM library provides several categories of features that enable verification IP to coordinate and cooperate, and enable reuse of verification IP. UVM provides a standard messaging mechanism that enables informational, warning, error, and fatal messages from all verification components to be tracked, filtered, and redirected. UVM provides a configuration mechanism that enables the operation of a verification IP component to be customized in a standardized way. UVM provides a phasing mechanism that enables verification IP components to coordinate and synchronize across the lifecycle of the test. In a UVM testbench, verification stimulus is modeled using a sequence, decoupling the type of stimulus to produce (the sequence) from the mechanism by which it is applied to the design. Finally, UVM provides the non-blocking analysis-port mechanism for broadcasting information about the state of the design or of activity within one of the verification IP components.

The UVM SDV package enables embedded software running on a processor model to take advantage of these UVM-provided services, enabling the embedded software to act as yet another verification component (or set of verification components) in the UVM testbench environment, coordinate with other activity occurring in the SystemVerilog portion of the UVM testbench, and publish information on the state of the software for use by verification IP components within the SystemVerilog portion of the testbench.

III. UVM SDV PACKAGE COMPONENTS

The UVM SDV package provides three key components, with a goal of simplifying the process of creating a software-driven verification environment.

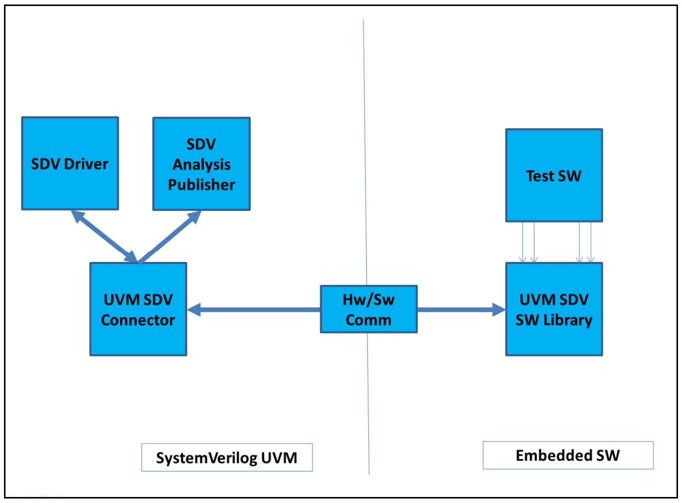

Figure 3 - UVM SDV Package Components

Figure 3 shows the components provided by the UVM SDV package in the SystemVerilog and embedded software portions of the verification environment. The UVM SDV connector manages interactions between the SystemVerilog environment and one processor. Consequently, an instance of the UVM SDV connector is created for each processor to be used for verification in the design. Communication between the SystemVerilog environment and the embedded software occurs via the hardware/software communication mechanism in the center of the diagram. The embedded test software uses the UVM SDV software library to interact with the UVM SDV connector, via the hardware/software communication mechanism.

A. Hw/Sw Communication Mechanism

The central enabling feature of the UVM SDV package is a communication channel between the SystemVerilog and the embedded software portions of the testbench environment.

Modularity is a key requirement for the communication mechanism, since the available infrastructure for communication between hardware and software may differ in a block-level environment, simulation-based system-level environment, emulation-based system-level environment, and a live target. Consequently, the UVM SDV package is architected to enable the communication mechanism to be easily swapped without requiring invasive changes to the embedded verification software or the SystemVerilog portion of the UVM environment.

Data transmitted via the communication mechanism is formatted as variable-length data packets, which enables specific implementations of the communication mechanism to know nothing about the higher-level activity that is provoking exchange of data. Consequently, creating a new implementation of the communication mechanism is relatively straightforward.

The UVM SDV package provides two pre-built communication mechanisms, which are suitable for simulation-based block-level verification, and simulation or emulation-based system-level verification.

Early software-driven verification can be performed at block-level by native-compiling the verification software as a DPI library that interacts with the block-level verification environment via the register model. The UVM C-Based Stimulus package [1] provides an implementation of this functionality. The UVM SDV package provides a DPI-based hardware/software communication mechanism for use in this type of environment.

In a simulation- or emulation-based system-level verification environment, data can be exchanged with the processor via a block of memory that is accessible to both the processor and the testbench environment. Many system-level environments already provide a mechanism to access processor-accessible memory from the testbench in order to load code or implement some form of software debug. The UVM SDV package provides a shared-memory communication mechanism that can easily be customized for use in a system-level verification environment.

New communication mechanisms can be created as needed. For example, a communication mechanism based on a high-speed serial interface could be useful in performing some level of software-driven verification on a live target.

B. Software Pack/Unpack API

Many of the application-level services provided by the UVM SDV package require the exchange of transactions between the SystemVerilog and the embedded software portions of the verification environment. UVM provides the uvm_packer class that enables a UVM object to serialize its contents to a data stream, or de-serialize its contents from a data stream.

The UVM SDV package provides a similar API for use by the embedded software. As with the UVM packer API, the UVM SDV API supports packing and unpacking integer, string, and object data types.

The figures below show an example of a data structure and the pack and unpack functions for that data structure. Figure 4 shows a simple C struct that we might use to exchage a pair of values between the embedded software testbench and the SystemVerilog environment. Figure 5 shows the pack function for the *sw_txn* data structure. The *uvm_sdv_pack_int* API is used to pack the 32-bit integer fields A and B. Figure 6 shows the unpack function for the *sw_txn* data structure. The *uvm_sdv_unpack_int* API is used to read values for the A and B fields from the packer.

```
typedef struct sw_txn_s {
  int A;
  int B;
} sw_txn;
```

Figure 4 - Example C Transaction

```
void sw_txn_pack(
  uvm_sdv_packer  *packer,
  void            *obj)
{
  sw_txn *txn = (sw_txn *)obj;
  uvm_sdv_pack_int(packer, txn->A, 32);
  uvm_sdv_pack_int(packer, txn->B, 32);
}
```

Figure 5 – Example Pack Function

```
void sw_txn_unpack(
  uvm_sdv_packer  *packer,
  void            *obj)
{
  sw_txn *txn = (sw_txn *)obj;
  txn->A = uvm_sdv_unpack_int(packer, 32);
  txn->B = uvm_sdv_unpack_int(packer, 32);
}
```

Figure 6 – Example Unpack Function

IV. UVM SDV APPLICATION-LEVEL SERVICES

The UVM SDV package provides a set of application-level services on top of the communication mechanism between the embedded software and the SystemVerilog UVM environment. These services extend UVM's features for configuration, messaging, synchronization, stimulus generation, and analysis to the embedded software environment.

A. *Access to the UVM Configuration Database*

The UVM configuration database enables the verification environment to configure instantiated verification components in a standard manner. The UVM configuration database enables data of any type to be stored and retrieved. Providing the embedded-software test access to the configuration database is highly desirable, since the operation of the software may need to adjust based on the design or environment configuration.

The UVM SDV package provides access to the configuration database for a restricted set of data types. Specifically, the API provides access to configuration items stored as a string, 64-bit integer, or uvm_object. A SystemVerilog class for a configuration item stored as an object must implement pack functionality, and a corresponding unpack function must be implemented in the software environment.

In the SystemVerilog environment, configuration elements intended for use by the embedded software are applied to the UVM SDV connector. Given the restricted configuration data types supported by the UVM SDV package, the configuration elements must be registered using one of pre-defined configuration types *uvm_sdv_config_type_obj, uvm_sdv_config_type_int,* or *uvm_sdv_config_type_str.*

```
typedef cfg_db_obj
 uvm_config_db #(uvm_sdv_config_type_obj);

function void build_phase();
  sw_cfg cfg_obj = new;

  cfg_obj.en_back2back = 1;
  cfg_obj.en_wide_link = 0;

  cfg_db_obj::set(this,
    “*.m_sdv_agent”,
    “TEST_SW_CONFIG”,
    cfg_obj);

endfunction
```

Figure 7 - Setting Configuration Data for SW

Figure 7 shows the SystemVerilog code to apply a configuration object for use by the embedded software to the UVM SDV connector. Figure 8 shows the corresponding embedded software to retrieve the configuration object and call SW APIs to apply the configuration.

```
void apply_test_config() {
  sw_cfg cfg_obj;

  uvm_sdv_config_db_get_object(
    “TEST_SW_CONFIG”,
    &sw_cfg_unpack,
    &cfg_obj);

  setup_back2back(cfg_obj.en_back2back);
  setup_wide_link(cfg_obj.en_wide_link);

}
```

Figure 8 - Obtaining Configuration Data in SW

B. *Unified Message Reporting*

UVM provides infrastructure for verification components to report messages. UVM SDV provides a corresponding API to the embedded software environment, enabling messages produced by software verification components to be managed in the same way that messages produced verification components in the SystemVerilog UVM environment are. This enables, for example, errors encountered by the embedded software to be counted in the simulation summary report along with errors encountered by any other verification component in the environment.

```
void test_func() {
  char buf[128];
  int A, B, i;

 for (i=0; i<4; i++) {
  A = read_A();
  B = read_B();

  sprintf(buf, “A=%d B=%d”, A, B);
  UVM_INFO(“test_func”, msg, UVM_MEDIUM);

  if (A >= B) {
    sprintf(buf, “A >= B”);
    UVM_ERROR(“test_func”, msg);
  }
 }
}
```

Figure 9 - Software Calling UVM Message API

Having messages from both SystemVerilog and embedded-software environments treated in the same way enables messages produced by the embedded-software environment to take advantage of any UVM-specific automation provided by the simulation environment. For example, the simulation environment may provide a message viewer that enables UVM messages to be categorized, or special filtering capability to search messages based on message-field attributes.

In addition to the benefit of unified message reporting, using the UVM SDV message API provides a performance boost compared to other simulation-based approaches to displaying messages. Often, a UART or other serial device is used to display messages. Even when the UART is configured for maximum baud rate, transmitting a message consumes a significant amount of simulation time. For example, displaying a short message via the UART in the UVM SDV demo testbench consumes 1600uS of simulation time. Reporting that same message via UVM SDV consumes 300uS of simulation time – a 5x reduction.

C. *Support for Objections*

UVM provides the objection mechanism to enable multiple processes to coordinate behavior. Often, the objection mechanism is used to coordinate the end of the main phase, and thus the end of the test.

The UVM SDV package extends the objection mechanism into the embedded-software domain, enabling the software test to participate in ending the test, as shown in Figure 10.

```
void test_main() {

  // Raise the start-of-test objection
  uvm_sdv_raise_objection(“Test Main”, 1);

  // Execute test behavior
  // ...

  // Raise the start-of-test objection
  uvm_sdv_drop_objection(“Test Main”, 1);
```

Figure 10 - Raising/Lowering an Objection

While not a complex feature, extending support for objections into the software domain makes it dramatically simpler to treat the embedded software as just another verification component in the UVM testbench environment.

D. *Software UVM Sequence Driver*

Stimulus in a UVM environment is commonly generated using a sequence running on a sequencer. The sequence generates sequence items that are supplied to a driver whose responsibility it is to convert the information within the sequence item into lower-level (often signal-level) interactions with the design. Figure 11 shows the UVM sequence, sequencer, and driver architecture for applying stimulus to the design.

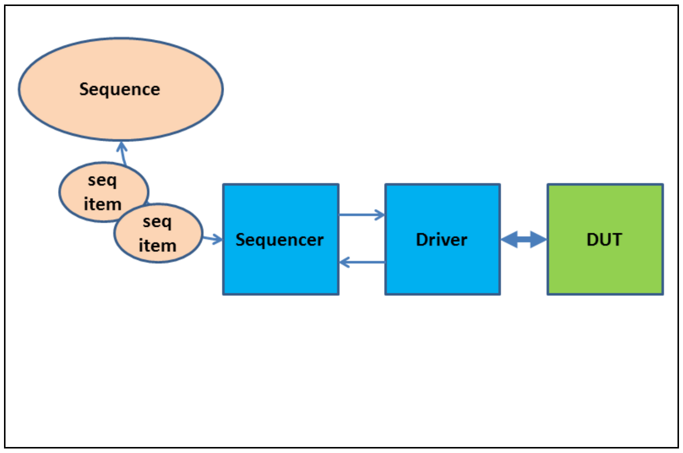

Figure 11 - UVM Sequence Architecture

The UVM SDV package extends the concept of a driver, enabling the embedded software to act as the driver for sequences running in the SystemVerilog environment. This enables stimulus described by sequences developed during block-level verification to be reused at system level to control the operation of the embedded software. It also enables the SystemVerilog portion of the testbench to coordinate operation of the software with stimulus generated via agents attached to design interfaces.

Most critically, enabling the embedded software to act as a driver for stimulus generated within the SystemVerilog environment enables software-driven verification to take advantage of advances in automated stimulus generation. Constrained-random stimulus generation boosts test-creation productivity significantly compared to directed tests, and has been used extensively for block-level verification. Leveraging block-level constrained-random sequences at the system level, via software-driven verification, boosts the number of use-case scenarios that can be described.

Efficiency, however, is a key concern for system-level verification. Given the achievable throughput at full-chip level, maximizing the number of unique cases verified and minimizing the number of redundant cases is highly desirable. Graph-based intelligent testbench automation tools, such as Mentor’s Questa inFact, can help achieve this goal by targeting desirable scenarios and eliminating unwanted redundancy. Since graph-based stimulus can integrate into a testbench as a UVM sequence producing sequence items, the same coverage-targeting graph-based sequences that drive verification IP in the SystemVerilog portion of the UVM environment can be used to drive embedded software via the UVM SDV sequence driver.

Figure 12 shows the UVM SDV architecture for applying sequence-driven stimulus via the embedded software. Note that the driver in the SystemVerilog UVM environment that formerly applied the stimulus to the design is replaced with a driver that forwards the stimulus, via the UVM SDV connector, to the embedded software.

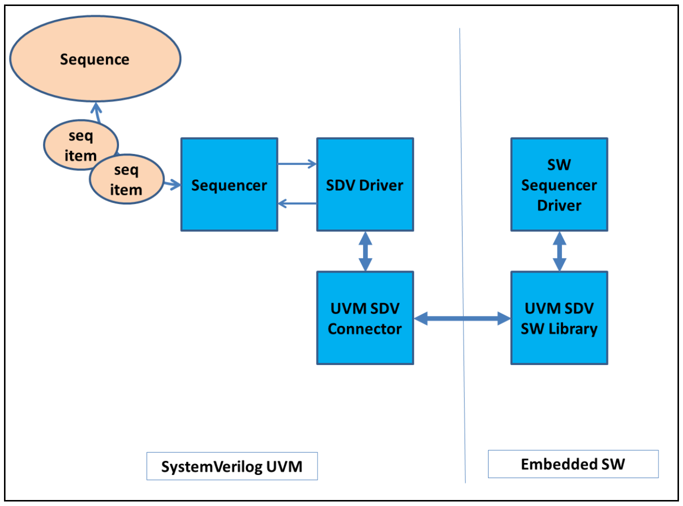

Figure 12 - SDV Sequencer Driver Architecture

Figure 13 shows the embedded software that implements a driver for a UVM sequencer. An instance of a sequencer driver is represented by a variable of type uvm_sdv_sequencer_driver_t, a data structure that encapsulates data required by the UVM SDV library to communicate with the SDV driver in the HVM environment. From the user's perspective, the uvm_sdv_sequencer_driver_t data structure is opaque. A handle for the sequencer driver is first initialized by specifying an instance-path pattern that matches the instance path of the UVM SDV driver within the SystemVerilog portion of the testbench, as well as the unpack function for the request item, and the pack function for the response item, if applicable. Once the sequencer driver is initialized, the software can begin calling the *get_next_item* function to receive a sequence item from the sequencer within the SystemVerilog portion of the environment. The *item_done* function is called to signal that the sequencer driver has completed the item and, optionally, return a response item.

```
req_txn req, rsp;
uvm_sdv_sequencer_driver_t txn_drv;

uvm_sdv_sequencer_driver_init(&txn_drv,
  "*.m_sdv_driver", // inst path of drv
  &req_txn_unpack,  // unpack function
  &rsp_txn_pack);   // pack function

while (true) {
 // Receive item from sequencer
 uvm_sdv_sequencer_driver_get_next_item(
    &txn_drv,
    &req);

 // Apply request and form response

 // Complete item
 uvm_sdv_sequencer_driver_item_done(
   &txn_drv,
   &rsp);
}
```

Figure 13 - Sequence Driver Software

E. *Launching Sequences*

Enabling the embedded software to implement a driver allows automated stimulus created in the SystemVerilog portion of the testbench environment to be easily applied via the embedded software. Often, though, it is also desirable for the embedded software to be able to easily initiate activity in the SystemVerilog portion of the testbench.

The UVM SDV package provides an API that enables the software to launch a sequence on any sequencer within the SystemVerilog portion of the environment, and check whether the sequence has completed. As shown in Figure 14, the software is able to create and start an arbitrary sequence on an arbitrary sequencer within the testbench environment.

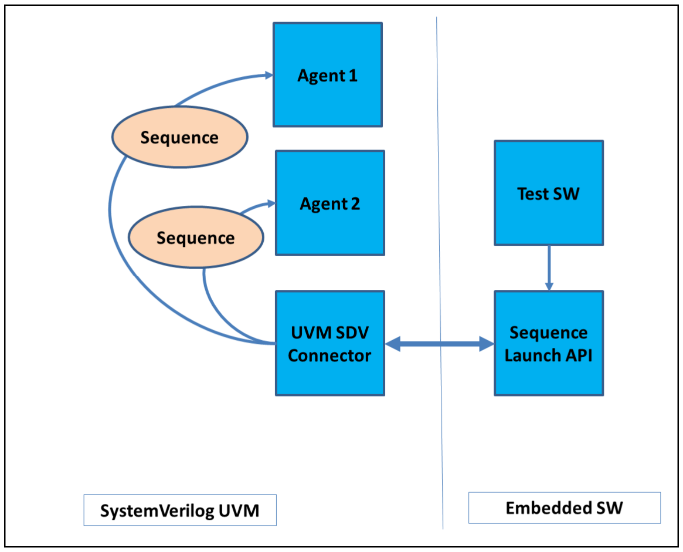

Figure 14 - Launching Sequences via UVM SDV

The code in Figure 15 shows an example application of the sequence-launch API. In this case, the embedded software launches a traffic-generation sequence in the SystemVerilog portion of the environment, and proceeds to run some data-processing code as long as the traffic is being generated by the sequence running in the SystemVerilog portion of the environment. This could be used to verify the ability of the design hardware to support both software-centric data processing and hardware-centric traffic.

```
uint32_t id; // sequence instance id

id = uvm_sdv_sequence_start(
    "traffic_gen_seq", // sequence id
    "*.traffic_seqr"); // sequencer path

// Perform other operations while
// the sequence is running
while (uvm_sdv_sequence_is_running(id)) {
  // Run software activity
  process_data();
}
```

Figure 15 - Sequence Launch Software

The ability of the embedded software to launch sequences within the SystemVerilog portion of the UVM environment, coupled with the ability for the embedded software to act as a driver for sequences, gives the software test maximum flexibility to act as a slave, a master, or both in the verification scenario.

F. Software Analysis Port

Analysis ports are widely used in UVM environments to publish data corresponding to events observed by the verification components or the accumulated internal state of the components. Analysis-port clients include scoreboards, functional coverage monitors, and sometimes complex stimulus-generation agents.

The UVM SDV package provides a method for embedded software to easily and efficiently send information to analysis subscribers in the SystemVerilog testbench environment. As shown in Figure 16, the UVM SDV package provides an analysis publisher component that is instantiated in the SystemVerilog testbench environment. This component contains an analysis port to which analysis subscribers in the environment can be connected. The UVM SDV embedded software library provides an API to establish a connection to an analysis publisher component, and send transactions to it.

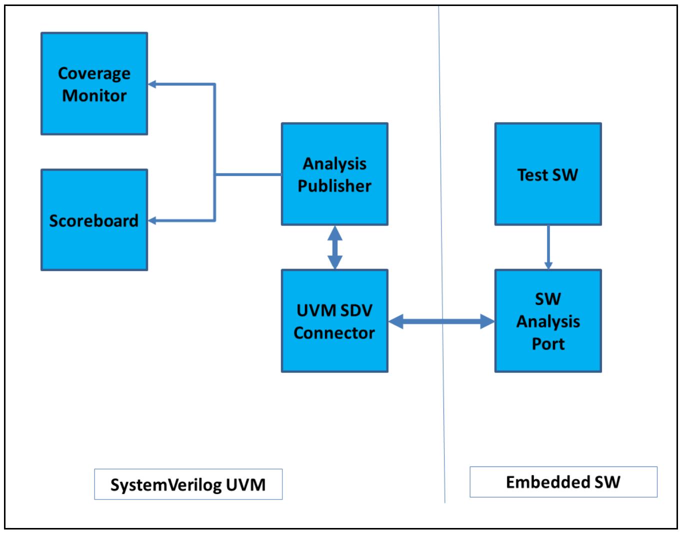

Figure 16 - SDV Analysis Port Architecture

Figure 17 shows the creation and connection of the Analysis Publisher component in the SystemVerilog portion of the testbench environment. A system-level scoreboard is connected to the analysis port of the analysis publisher component in order to monitor status information from the embedded software.

```
uvm_sdv_publisher #(sw_txn) m_state_pub;
system_scoreboard           m_sys_sb;

function void build();
  // Create the state publisher
  m_state_pub = new("m_state_pub", this);

  // Create the system scoreboard
  m_sys_sb = new("m_sys_sb", this);
endfunction

function void connect();
  // Connect the scoreboard to the
  // publisher analysis port
  m_state_pub.ap.connect(
    m_sys_sb.analysis_export);
endfunction
```

Figure 17 – Creation of the Analysis Publisher

Figure 18 shows the embedded software that initializes an analysis port to publish transactions via the analysis publisher component named *m_state_pub* instantiated in the SystemVerilog environment. In this case, each time the software test completes a set of operations, it publishes information (state *A* and *B*) to the SystemVerilog UVM environment.

```
uvm_sdv_analysis_port ap;
sw_txn state;

uvm_sdv_analysis_port_init(
  &ap,
  "*.m_state_pub",
  &sw_txn_pack);

while (true) {
 // Run operation
 process_data();

 // Read software state
 state.A = read_A();
 state.B = read_B();

 uvm_sdv_analysis_port_write(&ap, &state);
}
```

Figure 18 - Software Driving an Analysis Port

Efficiency is a key reason to use an analysis port to expose data for analysis from the embedded software environment. The Message Reporting section noted that using the UVM SDV messaging API to report a simple message was about 5x more efficient with respect to simulation time than transferring that same message via a UART. In the UVM SDV demo environment, transferring the same two elements of data using an analysis port consumes only 50uS of simulation time, compared to 300uS to display that information via the message-reporting services, and 1600uS to display the data via the UART. In other words, using the analysis port is 32x faster than displaying the data via a UART. In addition to shortening simulation times, more-efficient transfer of data minimizes the impact that analysis has on the timing and operation of the test.

G. *Extensibility to New Applications*

A key attribute of the UVM SDV package is its extensibility to new application-level services. New application-level services, whose requirements are not directly supported by existing services, can be implemented on top of the lower-level communication infrastructure provided by the UVM SDV package without modifying the package.

One example of an application-level services whose requirements differ from the support provided by existing services is reactive graph-based stimulus generation. Graph-based stimulus creating sequence items was mentioned earlier. However, the capabilities of graph-based stimulus generation are much more general and powerful than just creating transactions. Graphs lend themselves very nicely to the more-procedural nature of system-level scenarios, where a scenario may break down into a series of steps where decisions must be made at each step based on the current system state.

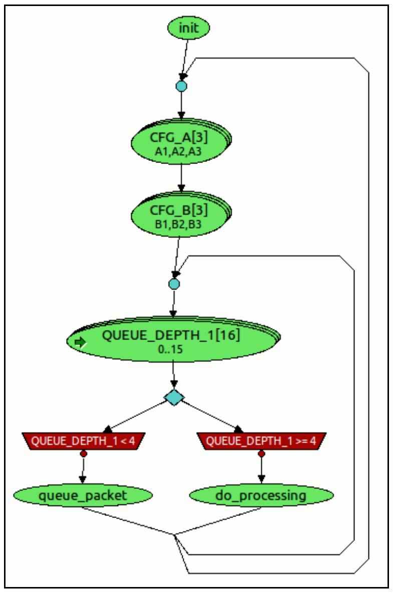

Figure 19 - Procedural Graph-Based Stimulus

Figure 19 shows a simple system-scenario graph where the system operation mode is configured (CFG_A, CFG_B), then a series of operations are performed. The current queue depth is queried at the beginning of each iteration of the lower loop in the graph. Depending on the queue depth, different operations are performed.

An application such as this requires both efficiency and support for exchange of heterogeneous messages between the embedded software and the SystemVerilog environment. Implementing the required exchange of information in terms of an existing UVM SDV application-level services doesn't make sense, but the underlying communication mechanism easily enables the flexible communication required by this application.

V. SUMMARY

Today's complex designs increasingly contain one or more processors. Leveraging these processors for software-driven design verification is critical to being able to exercise complex use-case scenarios. The UVM Software-Driven Verification package presented in this paper enables software-driven verification environments to easily be created. Extending UVM services such as automated stimulus generation, and efficient capture of analysis data to the software domain enables more-complex system use cases to be verified earlier and more-comprehensively. This bridging of the software and hardware worlds leads to increased productivity in system-level test creation, greater visibility into system-level interactions, and a boost in overall verification results.

REFERENCES

[1] M. Peryer, "C-Based Stimulus" [Online]. Available: https://verificationacademy.com/uvm-ovm/CBasedStimulus