Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

Multi Language Interpreter Embedding Tool for Shift Left Pre-Silicon Validation

ESTUDIO DE CASO para obtener el **GRADO** de **MAESTRO EN DISEÑO ELECTRÓNICO**

Presenta: **CHRISTIAN APARICIO ZULETA**

Director **CARLOS MARIO ANGULO PÁEZ**

Tlaquepaque, Jalisco. 18 de julio de 2019.

Acknowledgments

To my role-models, my parents Jorge and Alma, for your advices, your patience and for your guidance since day one, I feel fortunate of achieving this big personal and professional goal with you and my brother Erick by my side. Many thanks!

I am grateful to all the teachers at ITESO for sharing your knowledge with me and for your invaluable feedback that has boosted my career development, thank you all.

A special gratitude to all my coworkers at Intel Corporation, as each and every one of you motivates me to succeed, every personal achievement represents a milestone for the whole team and helps us consolidate as one, thanks for your encouragement.

To my manager and mentor, Carlos Angulo, for your trust on me and your excellent guidance for the past sixteen months, it has been an amazing experience.

Big thanks for your support, guidance and remarkable contributions to the people involved in the development of this project, Jonathan Newton for your inventiveness, patience and the great work done, Yaping Liu for sharing your knowledge with me and specially to Luke Chang, for your support and for playing a big bet on us, this is now a reality because of you. It has been a pleasure to share this stage of my life with you, keep inspiring those around you!

To my coworker, colleague and friend, Samed Abraham for your support, words of encouragement and for being there always looking after me, boosting my career with advices and long enriching talks.

Thank you all!

Abstract

Throughout the years, digital and analog designs have evolved meaningfully towards performance improvement, cost reduction and new features enablement. As a result, complexity has increased rapidly, demanding the development of better validation techniques in order to meet the time-to-market pressure calls with a bug free device. The primary choice of silicon development companies to validate software before the hardware becomes available, until now, is the FPGA based emulation platform, which leads to a big gap as it loads a register transfer level code that is usually not validated with SW-like flows in the early development stages. SW flows, mainly drivers, are validated in parallel to HW on SW emulation platforms. In order to fill the validation gap and push the finding of certain bugs to an earlier development stage, the idea of running SW tests with no or little modification in simulation environments would represent a big return of investment, rising the reliability of the system before manufacturing it, reducing time to market and development cost of the system on chip. This thesis explains the complete development of a framework able to run python scripts in VCS simulation by implementing the OVM Multi Language capability.

List of Figures

List of Tables

List of Acronyms, Abbreviations and Keywords

ASIC:	Application Specific Integrated Circuit
API:	Application programmers interface
Bug:	Design errors
CSR:	Control Status Register
DUT:	Design Under Test
FPGA:	Field-Programmable gate array
KVP:	Key Value Pair
Pre-silicon:	Usually associated with Validation before the silicon chip is available
RTL:	Register Transfer Level
SC:	System C
SoC:	System on Chip
SV:	System Verilog
TLM:	Transaction Level Modeling
UVM-ML:	Universal Verification Methodology Multi-Language
VC:	Verification Component
VCS:	Synopsis high performance simulation solution
VIP:	Validation Intellectual Property

Content

1. Introduction

Throughout the years, digital and analog designs have evolved meaningfully towards performance improvement, cost reduction and new features enablement. As a result, complexity has increased rapidly, demanding the development of better validation techniques in order to meet the time-to-market pressure calls with a bug free device.

The primary choice of silicon development companies to validate software (SW) before the hardware (HW) becomes available, until now, is the FPGA based emulation platform, which leads to a big gap as it loads a register transfer level (RTL) code that is usually not validated with SW-like flows in the early development stages. Performing SW validation on an emulation platform becomes a challenging team work as it requires the collaboration of both, HW and SW engineers solving first non RTL issues and then real design problems with limited visibility.

As the emulation platforms are used to validate SW flows before the silicon becomes available, the main goal of pre-silicon validation is to bring up individual features and connectivity, checking whether the implemented design meets defined specifications[1].

SW flows, mainly drivers, are usually validated in emulation platforms at the same time that HW is validated through pre-silicon simulations with real RTL. Even HW and SW validation efforts are essential activities in the integrated circuit design process, there is no way to detect discrepancy between both of them as the test content of both is completely different.

In order to fill this validation gap and push the finding of certain bugs to an earlier development stage, the idea of running SW tests with no or little modification in simulation environments would represent a big return of investment, rising the reliability of the system before manufacturing it, reducing time to market and development cost of the system on chip (SoC)[2]. While adding controllability, repeatability and observability to the behavior of the design when performing a pre-silicon simulation as stablished by Bob Bentley in the 47th Design Automation Conference [3].

The validation process of any robust SoC tends to be an iterative work, creating a cycle of run, debug, and fix several times until the design does what is meant to do with all the collaterals in place. It is common that a great amount of issues are from the verification environment or even errors while coding the tests. Every time a change is made to the code, it is needed to be compiled, this task can be as short as 5 minutes or as long as 1 or 2 hours, as a result, the compilation process slows down the validation progress.

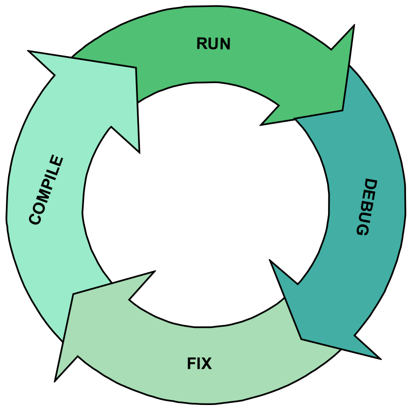

Figure 1-1: Traditional Validation Workcycle

By implementing the Multilanguage (ML) capability in an Universal Verification Methodology (UVM) environment, which is widely used all along the HW validation industry, for a specific Intellectual Property (IP) the integration of code written in different programming languages and methodologies can be achieved[4]. We call framework to every Validation Intellectual Property (VIP) implemented using a given language in such a way that UVM-ML can implement an internal communication between all of them and externally via its backplane adapters as seen in Figure 1-2. For the user, the backplane adapter is a normal Transaction Level Modeling (TLM) port, simple data transactions as integers or strings are copied from a source to a destination framework, while complex transactions like classes are serialized in the source end and then copied to the destination framework who is responsible of executing the deserialization to a matched type, finally events and barriers are synchronized with common events and barrier pools.

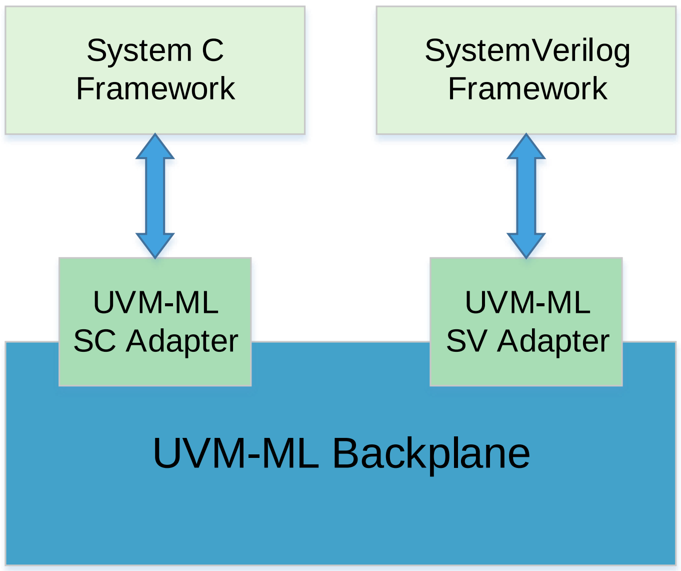

Figure 1-2: UVM-ML SC and SV Architecture

When having an UVM-ML environment that involves both, SystemVerilog (SV) and SystemC (SC), the SV framework usually operates as master, controlling simulation timing while the SC framework works as a slave synchronized to the master's time. In this side-by-side hierarchy, the UVM pre-run phases are executed individually in each framework while the run phases are executed synchronously.

All along the industry, nowadays Python is commonly chosen to write SoC SW tests that run in different software simulation platforms. The goal for the development of this thesis/project is to demonstrate that specific SW tests written in any interpreted language can stress a DUT during a pre-silicon simulation, pushing the finding of several potential design bugs earlier in the project timeline. This is achieved by introducing to the SC framework the capability of communicating with the desired interpreted language, Python in this case. The SC agent needs to launch an interpreter, which exchanges data with Python scripts and synchronizes with the rest of the UVM-ML components. A helper agent in the SV framework is also needed to process and generate according sequences as of the commands received form the SC framework, then the sequences, as in any UVM environment, drive stimulus to the Design Under Test (DUT) and finally return results back to the interpreter agent, leading to a bidirectional communication between Verification Components (VC).

An advantage of implementing python testing with a SC framework is that the testbench (TB) and sequences remain always the same and there is no need of recompiling every time a change is done to the test, as python is an interpreted language and the logic of it is extracted during runtime, reducing the time taken by the tradition validation work cycle.

The development of the complete Python Embedding Framework for pre-silicon simulations is discussed through this document, a brief background about the tools implemented is shown in Chapter 2, the high level overview and architecture are described in Chapter 3, it explains its architecture and the way it communicates through the UVM-ML backplane. Chapter 4 explains the main components needed in the testbench, Chapter 5 is about the way the scripts need to be written using the capabilities that the tool provides, meanwhile, Chapter 6 shows how the environment is built, and finally Chapter 7 provides details on the results obtained.

2. Background

This chapter explains the functionality of the foreign tools that make possible the end to end communication between Python (Interpeted language) and System Verilog (Hardware description and Verification language), the main component is the Multilanguage block, which enables the data exchange and synchronization between SV and C++ through a SC adapter, while Python talks to C++ through Boost and Ctypes functionalities.

2.1. UVM-ML

The Multi Language capability is an open source code that can be built in a Universal Verification Methodology environment. It forms the joint of multiple verification components (VC) implemented in different frameworks and syncs through the UVM-ML backplane. The environment can connect any amount of frameworks as long as every one of them has its own adapter properly connected to the UVM-ML backplane [5] as seen in Figure 2-1.

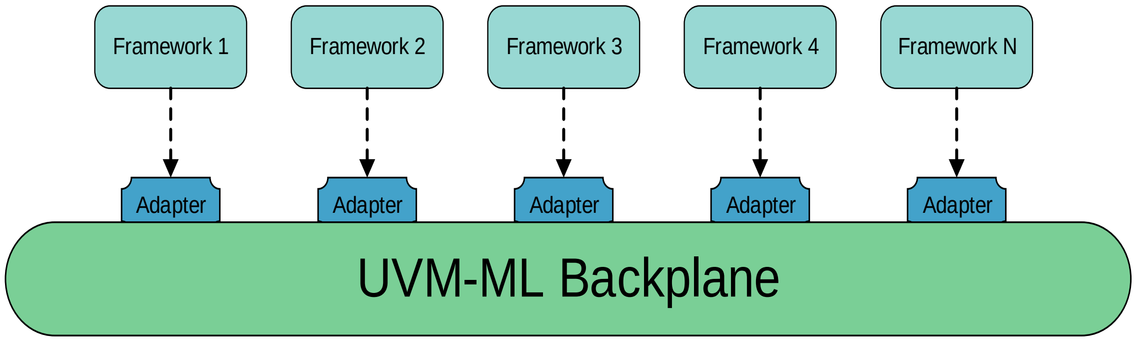

Figure 2-1: UVM-ML High level architecture

2.1.1 Adapters

Every framework adapter in the design must be registered in the backplane only once with a specific name. Then, a unique identifier is assigned to each adapter in order to enable communication with the backplane. The UVM-ML release contains the proper adapters needed for the development of this tool: SC and SV.

- The UVM-SV adapter is instantiated inside the uvm-ml package, for integrating it to the code the lines in Table 2-1 must be added:

```
import uvm_pkg::*;         // import the UVM-SV library package

`include "uvm_macros.svh" // macros which are part of the UVM-SV library

import uvm_ml::*;          // import the UVM-SV ML adapter (must come after the
above)
```

Table 2-1: UVM-SV adapter import

- The UVM-SC adapter is built on top of the UVM-SC included in the release, for integrating it to the code the lines in Table 2-2 must be added to system C code:

```
#include "uvm_ml.h"
using namespace uvm_ml;
```

Table 2-2: UVM-ML include statements in C++

2.2. Python extension to C++

Python programming language allows to add extension modules in order to be able to do things that usually cannot be achieved with python standalone. There is a Python application programmers interface (API) able to connect C or C++ code to Python, it can implement new built-in object types, call C library functions and do system calls[6].

This API, creates the connection between Python and C++, where then it is hooked to the UVM-ML backplane through the SC adapter. It consists of a set of functions, macros and variables accessing to Python run-time system and is incorporated in a C source file when including the header “Python.h”.

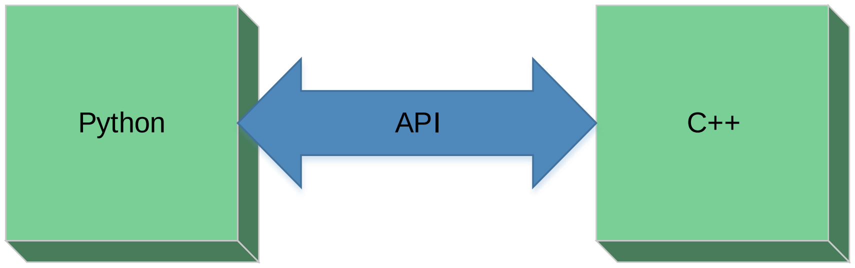

Figure 2-2: Python C++ extension

The Python interpreter must be compiled and linked with the Python system, this can be done dynamically or the C++ module can be set permanently part of the Python interpreter by adjusting its settings and rebuilding the Makefile.

This interface works for both, C++ function calls in Python and Python function calls in C++, this is mainly used for the libraries that support callback functions in such a way that a Python function could require calling a C++ callback function or the other way around. An example of how a Python function can be called in C++ can be seen in Appendix A. Python function call in C++.

Python interpreter is enhanced with third party tools such as ctypes [7] on the Python side and Boost [8] on the C++ side.

2.2.1 Ctypes

The foreign function library for Python called ctypes enables compatibility with C data types, making possible the data interaction between C programming language and Python. The functions defined in C can be executed from Python, nevertheless the arguments must be accurate as only data types none, integers, bytes object and strings can be used as parameters in C function calls from Python, while the remaining data types should be casted using ctypes functionality. The complete list of data types in both languages is shown in Table 2-3: C and Python datatypes mapping with ctypes.

Ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned longlong	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL terminated)	string or None
c_void_p	void *	int or None

Table 2-3: C and Python datatypes mapping with ctypes

2.2.2 Boost Python

On the other hand, Boost Python is a C++ library developed at Lawrence Berkeley National Laboratories, commonly used in the tests logic and implemented at the C++ interpreter to understand the packets formed at the python driver.

It exposes C++ classes, functions and objects to Python, and vice-versa through the C++ compiler. It supports:

- References and Pointers

Background

- Globally Registered Type Coercions
- Automatic Cross-Module Type Conversions
- Efficient Function Overloading
- C++ to Python Exception Translation
- Default Arguments
- Keyword Arguments
- Manipulating Python objects in C++
- Exporting C++ Iterators as Python Iterators
- Documentation Strings

The library functionality can be observed in Appendix B. Boost Python Hello World.

3. Architecture

3.1. Overview

The main benefits of implementing the Python Embedding Framework in a VCS (simulation tool) environment are:

- **Enable new content categories.** Software that will stress the DUT written in a language other than System Verilog can now run on the pre-silicon model. Resulting in a shift-left of the software activities, increasing confidence in the design as the software tools meant to run in a real platform can run on the pre-silicon design. As most of the post-silicon content is written in Python, the interpreter has been first created to embed Python over other interpreted languages.

- **Faster iteration of the content of the script.** Referring as a script to a test or a checker, a new iteration of it does not require any recompilation nor re-elaboration of the VCS model, it only demands the desired coding changes and rerunning the simulation. This saves a big amount of time per each iteration since the elaboration is not performed.

- **Lower cost of entry for writing validation content**. There is a high(er) cost of entry when someone new to the team needs to write checkers or other content for the validation tasks. A post-Si person coming to help out the pre-Si validation effort would normally need to come up to speed on System Verilog, OVM/UVM, the validation environment, etc. When implementing an interpreter, this person can now have a choice as to what language the new validation content is written in.

- **Cross validate full system simulator model with real RTL.** Usually in the silicon development process, software for a specific IP is created from the features and specifications defined by the architects and run in a full system simulator model such as a Simics[9] emulation model. With an interpreter tool, the real RTL can be cross validated with this model which results in having more reliable software.

- **Failure case portability.** When having equivalent models in different platforms and running the same test scenarios, the failures can be reproduced in any model, representing a big first step in debugging purposes.

This framework is able to inject stimulus to a DUT from a Python script, it is formed by:

- Python script/testcase
- Python Driver
- C++ interpreter
- UVM-ML SystemC Adapter
- UVM-ML SystemVerilog Adapter
- SystemVerilog Driver
- Traditional UVM environment

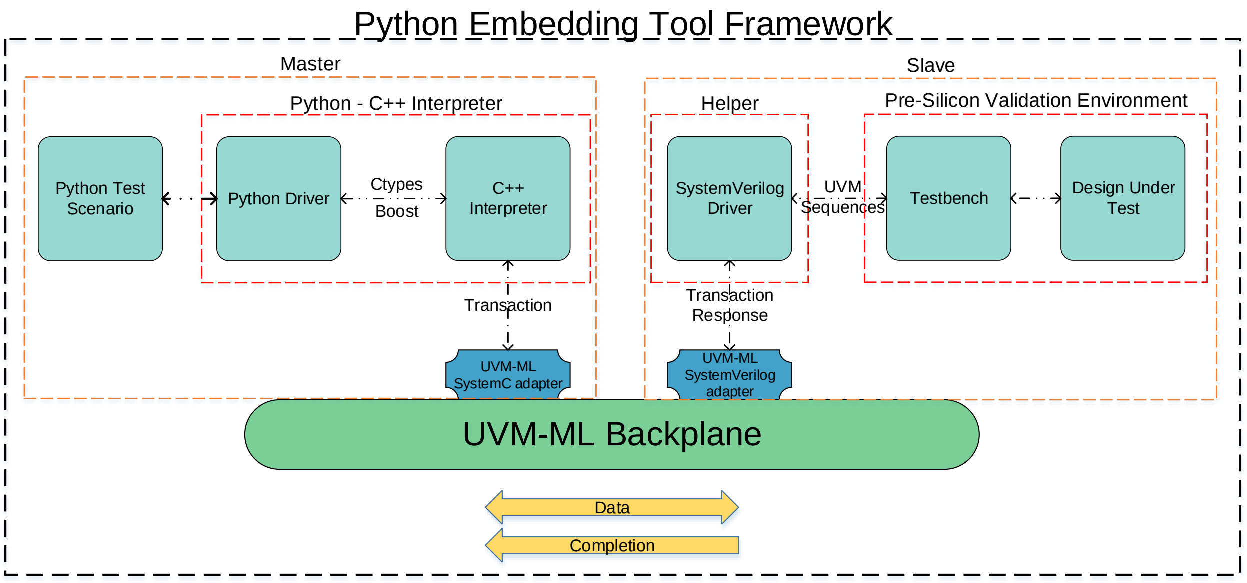

Figure 3-1: Python Embedding Tool Framework Block Diagram

Architecture

Once all the VCs are completely built and synchronized in the environment, the commands are initiated from the Python script, for example, a function call in a Python script that performs a control status register (CSR) read operation. First the Python interpreter agent forwards it to the UVM-ML SC adapter, then the UMV-ML backplane serializes the transaction and packs it into a byte stream. Afterwards, the SV framework adapter receives the byte stream, de-serializes it into matched transaction, the helper agent processes it, recognizes a CSR read command and generates the proper sequence in the pre-silicon validation environment. Upon the completion of the CSR read operation, the helper agent sends the response back to the Python script via the UMV-ML adapters and backplane.

3.2. Python – C++ Interpreter

3.2.1 Description

For this purpose, an interpreter is a validation component that connects to a VCS simulation via the Multilanguage capability provided by UVM. This tool can embed one or more interpreted language (Python for now) and allow scripts to run in a VCS simulation[10].

3.2.2 Architecture

The generic interpreter is primarily written in C++, using UVM-SystemC as the connection to the ML backplane, while the backplane itself connects to the System Verilog testbench and the specific language interpreters are embedded by using standard embedding tools as observed in Figure 3-2.

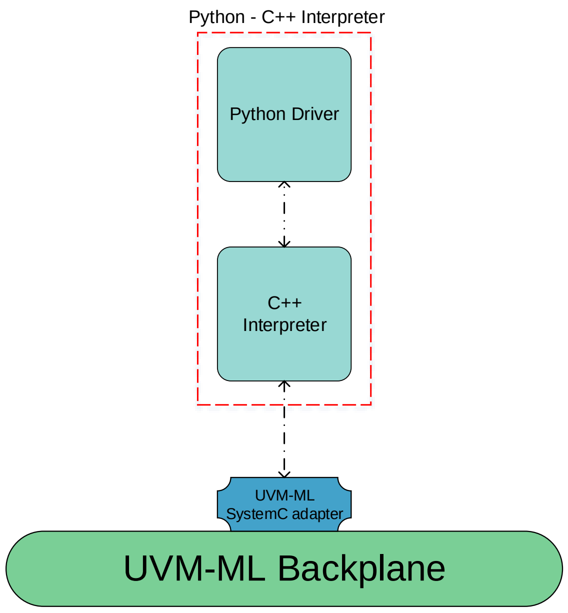

Figure 3-2: Python Interpreter High Level Architecture

As mentioned in section 2.2, the boost Python wrapper and ctypes functionality are used to create a bridge between the two main blocks of the interpreter, the driver which is written in Python and its C++ complement.

This tool has been developed with Python 3.x, so the Python interpreter and all its collaterals must be 3.x compliant.

3.2.3 Python Driver

Similar to SW drivers that provide SW applications functions to interact with HW devices, the Python driver in the interpreter side provides functions to the test for it to be able to interact with a pre-silicon validation environment. For a successful development, the driver must have two ends and work similar to a Master-Slave interface, where the Master is the Python Driver and has the control of the execution and the Slave is the C++ interpreter.

The logic and self-checking engines usually appear completely in Python, and the driver is the responsible of sending the control messages that eventually are going to be transformed into stimulus at the testbench. The driver functions are defined accordingly the needs of the project, but the typical functions for an application specific integrated circuit (ASIC) design are:

- **open():** Initializes the simulation environment and checks it's status to make sure it is ready to interact with tests and obtain resources such as memory allocations.

- **close():** releases the resources obtained by calling open().

- **csr_write():** CSR write operation with specified parameters sent as KVPs (address, data, etc). If the design is a PCIe device, this could be either CfgWrite or MMIO write.

- **csr_read():** CSR read operation with specified parameters sent as KVPs (address, number of bytes, etc.). If the design is a PCIe device, this could be either CfgRead or MMIO read.

- **mem_write():** host memory write operation. In a pre-silicon validation environment, this is usually a write to the system memory model. Certain address translation might also be performed if needed.

- **mem_read():** host memory read operation. In a pre-silicon validation environment, this is usually a read from the system memory model. Certain address translation might also be performed if needed.

- **sfence():** write fencing operation. For a PCIe design, this can be implemented as a zero-length MMIO read operation.

- **delay():** time delay operation.

3.3. Interpreter and Helper Communication

The bidirectional communication between an interpreter and a helper is achieved by fixed control messages with defined fields as in Table 3-2: Key Value Pairs (KVP) attributes for csr_read command, where the fields of name and rspReq are mandatory, the id is assigned automatically by the tool and any extra information can be sent through a key value pair. At the helper side, the messages are processed by the System Verilog driver and turned into sequences items returning completions status and any resulting data to the interpreter side, this can be observed in Figure 3-3.

The key/value pairs contained in the control messages are configurable depending on the needs of each function, in such a way that the interpreter only needs to understand these specific messages and not all the instructions delivered by the script. That is why a complement for the interpreter is needed at the System Verilog, which comprehends the model-specific commands, this component is called "helper".

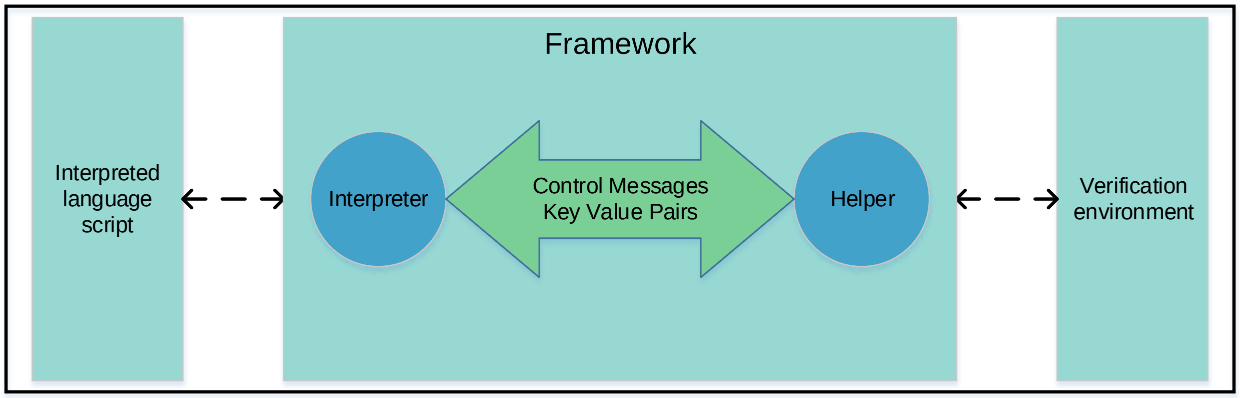

Figure 3-3: Interpreter and Helper Communication

3.3.1 Transaction

The transaction class is used as a container for the control messages sent between C++ and System Verilog, it is populated with 3 fixed variables and as many KVPs needed according to the function, these variables that are shown at the Table 3-1: Transaction class variables.

Field	Type	Description
id	int	Transaction ID, used to track transaction, maintained by driver
name	string	Transaction name such as "csr_read, mem_write", etc
rspReq	bool	Indicates whether a response is required
kvp	map<string, string>	Command specific attributes in <key, value> pairs

Table 3-1: Transaction class variables

The generic KVP allows any attributes of a command to be defined. For example, command "csr_read" definition is shown at Table 3-2: KVP attributes for csr_read command. The attributes of the remaining functions (Section 3.2.3) is not shown in this document, as their key value pairs might reveal some Intel Corporation confidential information.

Field	Type	Description
id	Auto	
name	"csr_read"	
rspReq	true	Requires response
kvp["addr"]	<string value>	CSR address in hex format
kvp["byte_count"]	<string value>	Number of bytes to read
kvp["result"]	<string value>	Data returned from read operation

Table 3-2: KVP attributes for csr_read command

The responses coming from the helper into the interpreter side are, also, generic KVP control messages with the fields shown in Table 3-3: Response KVP fields.

Field	Type	Description
kvp["Result"]	<string value>	Success or null
kvp["Info"]	<string value>	Extra information for debug purposes

Table 3-3: Response KVP fields

The C++ interpreter principal classes' definitions can be observed in Appendix C. Python driver engine C++ class definitions.

3.4. Helper Agent

The helper agent residing in the SV framework is like a typical UVM/OVM agent that contains a driver and a monitor. The monitor observes the SV adapter's TLM ports and forwards the transactions to the driver, at this point the received transactions are processed generating specific test sequences per command that are then transformed into stimulus by the pre-silicon validation environment. For the commands that require responses, the helper agent receives the results from the validation environment and sends back to the SC framework, where the interpreter agent completes a pending command.

The Python scripts can be executed once the transaction class is implemented in the interpreter and both drivers are added into the framework infrastructure. However, minor modification are necessary if the tests contain timing control such as calling time.sleep() function, running it in Python standalone the execution is paused during the timeframe desired, which does not match to the simulation timing source run by the master framework. These two separate timing sources cause undeterministic behavior of the simulation. As a solution, every time.sleep() function call must be replaced with the driver function delay(), which runs on the same timing source as the rest of the UVM-ML environment.

As an example, the implemented Python code in an application specific driver of the csr_write and csr_read is shown in Appendix D. Python driver code. A counterpart in SV with fixed response values was developed in order to test the functionality, this can be seen in Appendix E. SV development command processing. This last reference is the code used to validate the driver functionality and does not call any project specific sequence, it returns hardcoded values as a result.

4. Testbench Integration

The tool is integrated into the UVM compliant testbench by creating a new helper instance and added in the UVM database. The processCmd task must be called as it will handle all the commands that the script sends to the testbench. A good example of the implementation of the Multilanguage embedding tool in an UVM environment is presented in the code below, where the important items are **highlighted**.

```
…
  set_type_override_by_type(stl_intrp_embed_ctrl_helper::get_type(),
my_ctrl_helper::get_type());
  …
  class my_ctrl_helper extends stl_intrp_embed_ctrl_helper;
    `uvm_component_utils(my_ctrl_helper)

    // Standard UVM_Component functions
    function new(string name, uvm_component parent = null);
      super.new(name, parent);
    endfunction : new

    function void build();
      super.build();
    endfunction : build

    function void connect();
      super.connect();
    endfunction : connect
    /////////////
    // Main task - just pull stuff off the fifo and analyze the coverage sent
    task processCmd(stl_intrp_embed_ctrl_item ctrlIn, output bit processed);
      stl_intrp_embed_ctrl_send_one_seq   ctrl_send_seq;
      string idx;

      // see if base class handles this command, and if not we need to handle it
      processed = 0;
      super.processCmd(ctrlIn, processed);

      if (processed) begin
        `uvm_info(get_type_name(), "base class Processed Cmd from IntrpEmbed", UVM_FULL)
        return;
      end

      `uvm_info(get_type_name(), "Processing Cmd from IntrpEmbed - SandBox", UVM_FULL)

      if (ctrlIn.name == "CSR_READ") begin
        `uvm_info(get_type_name(), {"Got a CSR_READ command : ", ctrlIn.sprint}, UVM_NONE)
        ///////////////
        // Send the response – create a single-item sequence and set the results.
        ctrl_send_seq   =   stl_intrp_embed_ctrl_send_one_seq::type_id::create("cmd_response",
this);

     // it is critical you set the following 2 fields in this way.
     // Copy the ID and call it "command_response.
        ctrl_send_seq.ctrl.id             = ctrlIn.id;
        ctrl_send_seq.ctrl.name           = "command_response";

     // this is what the script should read as a result
        ctrl_send_seq.ctrl.kvp["result"]  = "98765";
```

```
// in this simple example, we copy all KVP items.  This is not needed, but you must set
       // kvp[STL_INTRP_EMBED_ENGINE_NAME] equal to the value from the command
     if (ctrlIn.kvp.first(idx) )
          do begin
            ctrl_send_seq.ctrl.kvp[idx] = ctrlIn.kvp[idx];
          end

          while (ctrlIn.kvp.next(idx));
        `ovm_info(get_type_name(),   {"sending   response   :   ",   ctrl_send_seq.ctrl.sprint},
OVM_NONE)

       // send the result
        ctrl_send_seq.start(sqr_h);

        // mark this as processed
        processed  = 1;
      end else begin
        // no - we didn't process!
        processed  = 0;

        `ovm_info(get_type_name(), {"unknown command: ", ctrlIn.sprint}, OVM_NONE)
      end
    endtask : processCmd
  endclass: my_ctrl_helper
```

Table 4-1: Interpreter tool UVM testbench

5. API

The interpreter tool has been designed to provide a set of primitives to make easier the debug and the sync with the testbench. These primitives are accessible from both, script and UVM sides in such a way that the information messages and errors will be triggered for both or in whichever side is needed. Depending on the project needs, new primitives can be defined, but for now the predefined ones are:

- **Writing to log file.** Equivalent to the UVM macro called `uvm_info the primitive LOGDebug() has been implemented and is used to display any information needed in the script and simulation logs during the execution.

- **Reporting errors.** The statements that use the LOGError() primitive will be reported as an error in the script execution and will be shown as a UVM_ERROR at the simulation log.

- **Reporting warnings**. The statements that use the LOGWarn() primitive will be reported as a warning in the script execution and will be shown as a UVM_WARNING at the simulation log.

- **Gating/Ungating quiescence.** In order to keep control of the execution, the set_quiesced() primitive can be set true to pause the simulation. This helps to have a deterministic system.

- **Subscription to a monitor.** The IE_Subscribe() primitive allows to subscribe to any monitor associated with a tracker.

- **Sending arbitrary commands.** The IE_Sendcmd() primitive allows to send any command to the helper instance that resides in the testbench.

5.1. C++/Python classes

By using the boost library in Python, the C++ classes get exposed to Python, allowing the execution of two fundamental classes: command and testbench interface.

5.1.1 Command Class – stl_intrp_embed_ctrl_py

The command class is a simple Key-Value Pair container that supports any command that the testbench helper will implement in the validation environment, the Table 5-1: Command class methods and properties shows its methods and properties.

Property/Method	Type	Description
constructor(string name)	method	Constructor to set the command name (see below)
name	string property	Sets the command name that will be used. "" by default, so it is needed to assign a value.
rspReq	Boolean property	Whether a response is required to this command. This indicates to the control helper that it must send an appropriate response back to the Python script. False by default.
id	unsigned int property	A unique ID assigned to this command. When instantiating this command, the ID is automatically assigned.

API

string getKVP(string key)	method	Gets the value associated with the given key.
setKVP(string key, string value)	method	Sets the value for the given key.
string toString()	method	Turns the command into a string for debug printout.
newID()	method	Assign a new ID to this command. When reusing a command instance, it needs to be assigned a newID before sending the command.

Table 5-1: Command class methods and properties

5.1.1.1 Example Usage

The following python code in Table 5-2 shows how a csr read command is created and filled with the proper fields.

```
# instantiate a cmd, name is “csr_read”
csrReadCmd  = stl_intrp_embed_ctrl_py("csr_read")
# this command requires a response
csrReadCmd.rspReq = True
# set the KVP fields associated with a CSR read
 csrReadCmd.setKVP("csr", "my_csr")
csrReadCmd.setKVP("field", "my_field")
csrReadCmd.setKVP("misc", "%d" %(1234))
```

Table 5-2:Python csr_read command

5.1.2 Testbench Interface – stl_intrp_embed_engine_python_py

This class represents the interface to the whole remaining simulation testbench. An instance of this is placed in the C++ as static python code in such a way that it is always available and it is not needed to be placed in every test case. The instance is done by placing the following line of code, while its methods and properties are described in Table 5-3: Testbench Interface class properties and methods.

```
stl_ie = stl_intrp_embed_engine_python_py()
```

Property / Method	Type	Description
release()	method	After doing initialization of the script and subscribing to any monitors, etc. This method must be called to enable the C++ side of things to progress. If it is not called, the simulation will be blocked from progressing. It is a way of synchronizing the Python script and the C++ side of things in a minimal way.
LOGInfo(string text)	method	Prints out the text to the script's log file (created by the C++ side of things). The text will be prepending by %I{Info}
LOGDebug(string text)	method	Prints out the text to the script's log file (created by the C++ side of things). The text will be prepending by %D{Debug}. This will be printed out only if debug is enabled in the STL IE.
LOGWarn(string text)	method	Prints out the text to the script's log file (created by the C++ side of things). The text will be prepending by %W{Warn} and will be sent to the UVM side of things as an uvm_warning.
reportError(string text)	method	Prints out the text to the script's log file (created by the C++ side of things). The text will be prepending by %E{Error} and will be sent to the UVM side of things as an uvm_error.
sendCmd(stl_intrp_embed_ctrl_py cmd)	method	Sends a command to the testbench.

API

stl_intrp_embed_ctrl_py getCmdResp(unsigned int id, bool wait)	method	Gets the result from the interpreter core for the command ID given. If the command has not received a result from the testbench, it will return with a KVP "result" equal to "NULL" and "info" equal to why. If wait is true, it will wait for the successful result. A successful result will have "result" equal to "SUCCESS" and the other KVPs set to the result from the testbench.
stl_intrp_embed_ctrl_py recvCmd()	method	Polls the C++ side of the interpreter to see if the testbench has sent a command to the script. The KVP "Empty" will be true if there is no command and false if there is. The name and other KVP items will be set if Empty is not true.
boolean hasRecvCmd()	method	Returns true if there is a command from the testbench waiting to be received by the script.
stl_intrp_embed_ctrl_py recvFeedback()	method	Polls the C++ side of the interpreter to see if the testbench has sent a feedback to a monitor that was subscribed to. The KVP "Empty" will be true if there is no feedback and false if there is. If there is feedback, then the KVP items will be set.
quiesced	Boolean property	Set this to True when the script determines that it is quiesced and False when it is not. It can be set and reset any time that the script determines this.

Table 5-3: Testbench Interface class properties and methods

5.1.2.1 Example Usage

The following Python code in Table 5-4 shows the usage of the methods provided by the testbench interface class:

```
import time

# start off by saying we're not quiesced
stl_ie.quiesced = False
###########
# release the c++ side of things
stl_ie.release()

count = 0
while (count < 10):
    if (count % 3 == 0):
        stl_ie.LOGInfo("count is %d \n" %(count))
    elif (count % 3 == 1):
        stl_ie.LOGDebug("count is %d \n" %(count))
    elif (count % 3 == 2):
        stl_ie.LOGWarn("count is %d \n" %(count))
    count = count + 1

stl_ie.LOGInfo("waiting 10 sec\n")
time.sleep(10)
stl_ie.LOGInfo("Done waiting\n")
# finish by indicating we are quiesced
stl_ie.quiesced = True
```

Table 5-4: Python base test

5.2. Standard Commands

Standard commands can be implemented by the tool, for now only the subscribe_to_monitor method is in place in the current tool release, its description can be seen in Table 5-5.

Command	Description
SUBSCRIBE_TO_MONITOR	Causes the testbench to subscribe to a monitor known by STL_SIMPLE_MON_SOURCE_GROUP and STL_SIMPLE_MON_SOURCE_INST, which are merged to create the "monitor_name" in the KVP. See example below. Those fields will be set in the returned values. Only monitors that are stl_simple_kvp_monitors work here since the connection to those monitors happens by default with the base helper class.

Table 5-5: Standard commands descriptions

5.2.1 Example Usage

The following Python code in Table 5-6 implements a Monitor Subscription:

```
import time
# start off by saying we're not quiesced
stl_ie.quiesced = False
stl_ie.LOGInfo("Doing Monitor Subscribe...\n")

# instantiate a cmd
subMonCmd  = stl_intrp_embed_ctrl_py("subscribe_to_monitor")
stl_ie.LOGInfo("Doing Monitor Subscription (ID :%d)\n" %(subMonCmd.id))
subMonCmd.setKVP("monitor_name", "sdt_sb_packet__bfm")

# send the command to the cpp side
stl_ie.sendCmd(subMonCmd)
###########
# release the c++ side of things
stl_ie.release()

# get the result
for i in range (0, 50):
    monFeedback = stl_ie.recvFeedback()
    stl_ie.LOGInfo("Waiting on Monitor Feedback #%d\n" %(i))
    while (monFeedback.getKVP("Empty") == "1"):
        stl_ie.LOGInfo("Got Monitor Feedback - Empty\n%s\n" %(monFeedback.toString()))
        time.sleep(1)
        monFeedback = stl_ie.recvFeedback()
    stl_ie.LOGInfo("Got Monitor Feedback\n%s\n" %(monFeedback.toString()))

stl_ie.LOGInfo("waiting 5 sec\n")
time.sleep(5)
stl_ie.LOGInfo("Done waiting\n")

# finish by indicating we are quiescedstl_ie.quiesced = True
```

Table 5-6: Python Monitor Subscription

The output of above code is seen in Table 5-7:

```
%I{INFO }{Time        295}{my_monitor_subscribe}:: PY: Got Monitor Feedback
Name:  MONITOR_FEEDBACK
Id:    18446744073709551516
rspReq:0
kvp[DESTLID] = 0xab041e
kvp[DIRECTION] = Tx
kvp[EMPTY] = 0
kvp[OPCODE] = SDT_SB_OP_RSP_E2E_ONLY_ACK
kvp[PACKETTYPE] = SDT_SB_TYPE_RSP
kvp[SOURCELID] = 0xd269f4
kvp[STL_INTRP_EMBED_ENGINE_NAME] = my_monitor_subscribe
kvp[STL_SIMPLE_MON_SOURCE_GROUP] = sdt_sb_packet
kvp[STL_SIMPLE_MON_SOURCE_INST] = bfm
kvp[TIME] =            59000
```

Table 5-7: Monitor Suscription Output

6. Environment setup

A basic driver test has been placed inside the Python Embedding Framework repository, this driver test works as a hello world and is able to demonstrate the robustness of the project, also helps debugging any issue on the functions implementation before merging it into any VIP environment.

Before cloning the embedded repository it is essential to have in place a working environment that includes the synopsis VCS simulation tool and Python 3 or greater.

The following section presents a step by step guide to run the python driver test with the UVM-ML capability in the VCS simulator.

6.1. Step by step execution guide

As there is a fixed infrastructure, the process of setup has been simplified into 3 main scripts that run in a linux shell, the first step is to generate the TDIF files from an XMLand compile them to enable the SV and C++ code for Multilanguage capability. This can be obtained by running the command in Table 6-1.

```
>> source scripts/make_tdif.cmd
```

Table 6-1: Multilanguage compilation command line

The expected output in the terminal is what can be seen in Table 6-2:

```
-I-: Generated code from XML
-I-: Source XML: /nfs/sc/disks/mst_pe_czuleta/stl_intrp_embed-
srvr10nm/stl_intrp_embed_common_pkg/tdif/ctrl.xml
-I-: Reading: /nfs/sc/disks/mst_pe_czuleta/stl_intrp_embed-
srvr10nm/stl_intrp_embed_common_pkg/tdif/ctrl.xml
-I-: Processing TDIF: Source: manual Version: 1
-I-: PARSING: class: stl_intrp_embed_ml_ctrl
-I-: Generating SV Code
-I-: Generating SC Code
-I-: Generating e Code
-I-: Done Generating Code
-I-: Done.
STL IE Regression PASSED!!
Test PASS
```

Table 6-2: Multilanguage compilation expected output

This script should be executed only once, as the Multilanguage plane code is already robust and is meant to be static. The next step is to run the script responsible of compiling the whole C++ infrastructure, every time a modification is made to the interpreter C++ code, the linux command line in Table 6-3 should be executed:

```
>> source scripts/make_sc.cmd
```

Table 6-3: C++ compilation command line

The expected output in the terminal is shown in Table 6-4:

```
scons: done building targets.
```

Table 6-4: C++ compilation expected output

Finally, the UVM testbench should be compiled after any modification made to the C++ or SV code, also this script executes as sanity, the base test for the interpreter tool with the command in Table 6-5.

```
>> source scripts/make_sandbox_full_ie.cmd
```

Table 6-5: UVM testbench compilation command line

The expected output in the terminal is seen in Table 6-6:

```
Compile PASSED for /nfs/…/sandbox/sv/full_ie
Simulation PASSED for /nfs/…/sandbox/sv/full_ie

STL IE Regression PASSED!!
```

Table 6-6: UVM testbench compilation expected output

7. Results

7.1. Base Test

As stated before, the base test runs by sourcing the full_ie makefile after compiling the SV code, the base test at the UVM side called "stl_intrp_embed_test_base" is executed, it is responsible of parsing the script name and the parameters needed for the VCS simulation, the command formed by the Makefile for running the base test is:

```
>> /nfs/…/sandbox/sv/full_ie/results_stl_intrp_embed_pkg_sles11/simv
+UVM_NUM_ERROR=1 +UVM_VERBOSITY=UVM_FULL +ntb_random_seed=1 +SC_ENVNAME=my_sc_env
+UVM_TESTNAME=stl_intrp_embed_test_base +stl_ie_debug=on +stl_ie_log=on
+stl_ie_tracker=on +stl_ie_min_quiesce_length=1001
+stl_ie_stop_count_threshold=6001 +stl_tracker_all=on +verbose=0
+stl_ie_script0=sandbox/sv/full_ie/myBaseTest.py +stl_ie_script_type0=python
+stl_ie_script_name0=my_python +stl_ie_script_param0="-bob='hello world'"
```

Table 7-1: Base test simulation command

On the other hand, the Python base test perform the primitives shown in Table 7-3: Testbench Interface class properties and methods, proving that the Python logic can be translated into UVM sequences, functions or tasks. The Python base test does not need to be included in every test scenario, hence, some lines should be legacy code for every Python simulation. The complete Python base test code is shown below, where the legacy code is **highlighted**.

```
# start off by saying we're not quiesced
ie_setQuiesced(False)
# release the c++ side of things
ie_release()
# Get the arguments
import sys
ie_LOGInfo("argv = %s\n"%(str(sys.argv)))
print(sys.argv)
# loop with some different print outs
count = 0
while (count < 10):
    if (count % 3 == 0):
        ie_LOGInfo("remainder 0, count is %d \n" %(count))
    elif (count % 3 == 1):
        ie_LOGDebug("remainder 1, count is %d \n" %(count))
    elif (count % 3 == 2):
        ie_LOGWarn("remainder 2, count is %d \n" %(count))
    count = count + 1
driver_sleep(.00000001)
ie_LOGInfo("Done waiting\n")
# finish by indicating we are quiesced
ie_setQuiesced(True)
```

Table 7-2: Python base test code

For every simulation performed, it is expected to have two separate log files, one is for the script execution and the remaining one is for the VCS simulation, which also is set to include the information displayed in the first log. The python run log of the base test prints the following statements.

```
%I{INFO }{Time 2}{my_python}:: spawning thread using script
sandbox/sv/full_ie/myBaseTest.py...
%I{INFO }{Time 2}{my_python}:: Waiting for CV...
%I{INFO }{Time 2}{my_python}:: Initializing Python
%I{INFO }{Time 2}{my_python}:: Adding arguments to argc/argv for Python
py_argv[0] = 0x10f69e0 (From /nfs/.../sandbox/sv/full_ie/myBaseTest.py)
py_argv[3] = 0x104abb0 (From -bob=hello world)

%I{INFO }{Time 2}{my_python}:: calling extract
%I{INFO }{Time 2}{my_python}:: extracted Python gbl_stl_ie object
%I{INFO }{Time 2}{my_python}:: executing file "sandbox/sv/full_ie/myBaseTest.py"
from path "/nfs/.../"
%I{INFO }{Time 2}{my_python}:: PY: LOG DEBUG. Loading VCS Driver
%I{INFO }{Time 2}{my_python}:: Released CV
%I{INFO }{Time 2}{my_python}:: Done waiting for CV...
%I{INFO }{Time 2}{my_python}:: PY: argv =
['/nfs/.../sandbox/sv/full_ie/myBaseTest.py', '-bob=hello world']
%I{INFO }{Time 2}{my_python}:: PY: remainder 0, count is 0
%D{DEBUG}{Time 2}{my_python}{T#     1}:: PY: remainder 1, count is 1
%W{WARN }{Time 2}{my_python}:: PY: remainder 2, count is 2
%I{INFO }{Time 2}{my_python}:: PY: remainder 0, count is 3
%D{DEBUG}{Time 2}{my_python}{T#     1}:: PY: remainder 1, count is 4
%W{WARN }{Time 2}{my_python}:: PY: remainder 2, count is 5
%I{INFO }{Time 2}{my_python}:: PY: remainder 0, count is 6
%D{DEBUG}{Time 2}{my_python}{T#     1}:: PY: remainder 1, count is 7
%W{WARN }{Time 2}{my_python}:: PY: remainder 2, count is 8
%I{INFO }{Time 2}{my_python}:: PY: remainder 0, count is 9
%I{INFO }{Time 2}{my_python}:: PY: waiting 10 ticks
%I{INFO }{Time 2}{my_python}:: PY: DELAY n = 1 ticks
%I{INFO }{Time 3}{my_python}:: PY: Done waiting
%I{INFO }{Time 3}{my_python}:: python exec_file success!
```

Table 7-3: Base test Python simulation log

The VCS simulation log is where every script, as well as, logic design or verification component messages are merged, in the specific case of the Python base test, the interpreter is the only component in the environment, so it is expected to see the same messages that in the interpreter but with a timestamp and UVM format, as seen below.

Results

```
UVM_INFO  @ 0: reporter [UVM-ML]: Found framework command line arg
UVM_TESTNAME=stl_intrp_embed_test_base
SC_ENVNAME=my_sc_env

UVM_INFO /.../tb_comp_pkg.svh(290)@0: [my_ctrl_subscriber] New component created
UVM_INFO /.../tb_comp_pkg.svh(422)@0: [IE Sandbox Test] Checking for STL IE SB
Plusargs

UVM_INFO /.../tb_comp_pkg.svh(56)@0: [stl_ie_sb_env] build called

SC intrp_embed_env::end_of_elaboration
SC intrp_embed_env::start_of_simulation

UVM_INFO /.../tb_comp_pkg.svh(454)@0:[TEST] SV run phase

UVM_INFO /.../tb_comp_pkg.svh(498)@0:[stl_intrp_embed_test_base] Tick           0

UVM_INFO /.../stl_intrp_embed_ctrl_mon.sv(56)@2000: [ie_ctrl_mon] STL INTRP EMBED
is quiesced
UVM_INFO /.../stl_intrp_embed_ctrl_mon.sv(60)@3000: [ie_ctrl_mon] STL INTRP EMBED
is not quiesced

UVM_INFO /.../stl_intrp_embed_ctrl_mon.sv(47)@4000: [STL INTRP EMBED] {my_python}::
PY: remainder 0, count is 0

UVM_INFO /.../stl_intrp_embed_ctrl_mon.sv(47)@4000: [DEBUG] {my_python}:: PY:
remainder 1, count is 1

UVM_WARNING /.../stl_intrp_embed_ctrl_mon.sv(49)@4000: [STL INTRP EMBED]
{my_python}:: PY: remainder 2, count is 2


--- UVM Report Summary ---

Quit count :           0 of           1
** Report counts by severity
UVM_INFO :  251
UVM_WARNING :   10
UVM_ERROR :    0
UVM_FATAL :    0
```

Table 7-4: Base test VCS simulation log

7.2. Driver Test

The driver test implements in a sandbox area all the functions that have been defined in the Python driver (Section 3.2.3), the source code is shown next.

```
import time
import imp
import os
import subprocess
import sys

ie_setQuiesced(False)
ie_release()

open_hdl()

wr_csr(0x0,0x4e31)
wr_config_csr(0,0x4e31)
rd_csr(0x1e01010)
rd_config_csr(0x0)
driver_sleep(.000001)
sfence()
wr_csr(0x8,0x4e31)
sfence()
wr_csr(0x402008,0x4e31)
sfence()
rd_csr(0xc04008)
wr_csr(0xc04008,0x4e31)
sfence()

wr_host_mem(8,0x1e01010,[0x4A])
wr_host_mem8b(0x1e01010,[0x1A2B3C4D5E6F7A8B])
wr_host_mem4b(0x1e01010,[0x1A2B3C4D5E6F7A8B])
wr_host_mem2b(0x1e01010,[0x1A2B3C4D5E6F7A8B])
wr_host_mem1b(0x1e01010,[0x1A2B3C4D5E6F7A8B])

ie_LOGInfo("LOG DEBUG. Read host mem return %s\n"%rd_host_mem(8,0x1e01010))
ie_LOGInfo("LOG DEBUG. Read host mem return %s\n"%rd_host_mem8b(0x1e01010))
ie_LOGInfo("LOG DEBUG. Read host mem return %s\n"%rd_host_mem4b(0x1e01010))
ie_LOGInfo("LOG DEBUG. Read host mem return %s\n"%rd_host_mem2b(0x1e01010))
delay(10)
close_hdl()
ie_LOGInfo("LOG DEBUG. Close return handle value:%d\n"%(fxr_hdl))
ie_setQuiesced(True)
```

Table 7-5: Driver Test source code

Results

After running the simulation the Python logfile shows initialization, the parsing of arguments, the loading of the proper drivers, the sends and responses for each command (Just the first CSR_WRITE shown here) and finally the test overall result:

```
%I{INFO }{Time   2}{my_driver_test}:: spawning thread using script
sandbox/sv/driver_test/driver_test.py...

%I{INFO }{Time   2}{my_driver_test}:: Waiting for CV...
%I{INFO }{Time   2}{my_driver_test}:: Initializing Python
%I{INFO }{Time   2}{my_driver_test}:: Adding arguments to argc/argv for Python
argc = 1 argv =

%I{INFO }{Time   2}{my_driver_test}:: PY: LOG DEBUG. Loading VCS Driver
%I{INFO }{Time   2}{my_driver_test}:: PY: LOG DEBUG. Test import driver
%I{INFO }{Time   2}{my_driver_test}:: Released CV

%I{INFO }{Time   7}{my_driver_test}:: Calling sendCmd with cmd info: id:1 (new: 1)
name: csr_write (new: CSR_WRITE)
%D{DEBUG}{Time   7}{my_driver_test}{T#     6}:: inserted cmdID 1

%D{DEBUG}{Time  12}{my_driver_test}{T#    11}:: in getCmdResp - cmdId = 1 got a
response
retVal = SystemC Class Data Dumper for Class stl_intrp_embed_ctrl
                          name = COMMAND_RESPONSE
                            id = 0x00000001
                       rspReq? = false
                  kvp[ADDR] = "0x0"
            kvp[BYTE_COUNT] = "8"
              kvp[CSR_NAME] = "CSR[0][0]"
                  kvp[DATA] = "0x4e31"
                kvp[RESULT] = "1"
kvp[STL_INTRP_EMBED_ENGINE_NAME] = "my_driver_test"


%I{INFO }{Time 123}{my_driver_test}:: PY: CLOSE HDL. Handler has been closed
successfully

%I{INFO }{Time 123}{my_driver_test}:: PY: LOG DEBUG. Close return handle value:-1
%I{INFO }{Time 123}{my_driver_test}:: python exec_file success!
```

Table 7-6: Python Driver Test Python log

The equivalent information is shown in the VCS log file, the info, debug or even messages of this log are created by the testbench after being called from python, while in the previous log are created fully in Python:

```
UVM_INFO /nfs/.../sandbox/sv/full_ie//tb_comp_pkg.svh(161) @ 10000:
stl_intrp_embed_test_base.stl_ie_sb_env.ie_ctrl.ie_ctrl_help [my_ctrl_helper] Got a
CSR_WRITE command :
Name                      Type                Size                Value
-----------------------------------------------------------------------
stl_intrp_embed_common_+ stl_intrp_embed_ct+ -
stl_intrp_embed_common_pkg::stl_intrp_embed_ctrl@321
  name                    string              9               CSR_WRITE
  id                      integral            64                    'h1
  rspReq                  integral            1                     'h1
  kvp                     aa(string,string)   6                       -
    [ADDR]                string              3                     0x0
    [BYTE_COUNT]          string              1                       8
    [CSR_NAME]            string              17              CSR[0][0]
    [DATA]                string              6                  0x4e31
    [STL_INTRP_EMBED_EN+ string              14         my_driver_test
-----------------------------------------------------------------------

UVM_INFO /nfs/.../sandbox/sv/full_ie//tb_comp_pkg.svh(175) @ 10000:
stl_intrp_embed_test_base.stl_ie_sb_env.ie_ctrl.ie_ctrl_help [my_ctrl_helper]
sending response :
Name                      Type                Size                Value
-----------------------------------------------------------------------
ctrl_send_one_seq_ctrl    stl_intrp_embed_ct+ -    ctrl_send_one_seq_ctrl@325
  name                    string              16       command_response
  id                      integral            64                    'h1
  rspReq                  integral            1                     'h0
  kvp                     aa(string,string)   7                       -
    [ADDR]                string              3                     0x0
    [BYTE_COUNT]          string              1                       8
    [CSR_NAME]            string              17              CSR[0][0]
    [DATA]                string              6                  0x4e31
    [STL_INTRP_EMBED_EN+ string              14         my_driver_test
    [result]              string              1                       1
-----------------------------------------------------------------------
…
UVM_INFO /nfs/.../sandbox/sv/full_ie//tb_comp_pkg.svh(443) @ 40156000:
stl_intrp_embed_test_base [Test PASS ] SV report phase

--- UVM Report Summary ---

Quit count :           0 of           1
** Report counts by severity
UVM_INFO :  371
UVM_WARNING :    9
UVM_ERROR :    0
UVM_FATAL :    0
```

Table 7-7: Python Driver Test VCS log

Results

Currently, in this development infrastructure, there is not a DUT instance in place, thus there are not interfaces where the stimulus can be observed in waveforms. Nevertheless, it has been demonstrated that Python can be understood by an UVM testbench, where it is easily transformed into stimulus.

8. Conclusion

Nowadays, in the changing ecosystem of the integrated circuits design process, the FPGA based emulation platforms are commonly used to enable SW/HW co-validation before the specific platform is available. This project demonstrates robust and reliable validation component that represents an alternative or a complement to FPGA platforms with a valuable advantage of mapping real chip usage code directives into a simulation model, by integrating it successfully, this tool is able to reproduce any post-silicon issue in simulation model, or even find it earlier in the project development while decreasing its cost, increasing its quality and simplifying the debug process.

The Multilanguage Interpreter Embedding tool is part of a process of continuous improvement inside Intel Corporation, its development represents a major effort to deliver high quality products to market as the idea was conceived after detecting a gap within the SOC development process between the design verification and the post-silicon validation process.

This Python Interpreter Embedding tool for shift-left validation has been successfully implemented within an IP at Intel Corporation, leading to the first time a SW test runs in a pre-silicon model and supporting the advantages this thesis establishes.

At this point, it has been demonstrated that interpreted language scripts can be executed in a pre-silicon simulation model fulfilling its objectives of cross validating software with hardware and accelerating the validation process.

Future work involves the integration to a specific IP model, create its own driver and obtain a benchmark out of its behavior. Also, adding the support of other scripting languages such as Perl or Ruby should be considered for the continuity of this project.

Appendix A. Python function call in C

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

Appendix B. Boost Python Hello World

C++ function definition:

```
char const* greet()
{
   return "hello, world";
}
```

Python call:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

Python run:

```
>>  import hello_ext
>>> print hello_ext.greet()
hello, world
```

Appendix C. Python driver engine C++ class definitions

```
BOOST_PYTHON_MODULE(em_stl_intrp_embed_ctrl_py) {
  python::class_<stl_intrp_embed_ctrl_py> ("stl_intrp_embed_ctrl_py",
init<std::string>())
    .def("getKVP",               &stl_intrp_embed_ctrl_py::getKVP)
    .def("setKVP",               &stl_intrp_embed_ctrl_py::setKVP)
    .def("toString",             &stl_intrp_embed_ctrl_py::toString)
    .def("newID",                &stl_intrp_embed_ctrl_py::newID)
    .add_property("name",        &stl_intrp_embed_ctrl_py::getName,
&stl_intrp_embed_ctrl_py::setName)
    .add_property("rspReq",      &stl_intrp_embed_ctrl_py::getRspReq,
&stl_intrp_embed_ctrl_py::setRspReq)
    .add_property("id",          &stl_intrp_embed_ctrl_py::getID,
&stl_intrp_embed_ctrl_py::setID);
}

void stl_intrp_embed_ctrl_py::setKVP(string key, string val) {
  string localKey = key;
  strToUpper(localKey);
  kvp[localKey] = val;
}

string stl_intrp_embed_ctrl_py::toString() const {
  stringstream ss;

  ss << "Name:  " << name << endl;
  ss << "Id:    " << id << endl;
  ss << "rspReq:" << rspReq << endl;
  for (auto itr = kvp.begin(); itr != kvp.end(); ++itr) {
    ss << "kvp[" << itr->first << "] = " << itr->second << endl;
  }

  return ss.str();
}

string stl_intrp_embed_ctrl_py::getKVP(string key) {
  string localKey = key;
  strToUpper(localKey);

  auto itr = kvp.find(localKey);
  if (itr != kvp.end()) {
    return itr->second;
  } else {
    return string("");
  }
}


// Pack the Base class wrapper into a module
BOOST_PYTHON_MODULE(em_stl_intrp_embed_engine_python_py) {
  python::class_<stl_intrp_embed_engine_python_py>
("stl_intrp_embed_engine_python_py")
    .def("LOGInfo",              &stl_intrp_embed_engine_python_py::LOGInfo)
    .def("LOGWarn",              &stl_intrp_embed_engine_python_py::LOGWarn)
    .def("reportError",          &stl_intrp_embed_engine_python_py::reportError)
    .def("LOGDebug",             &stl_intrp_embed_engine_python_py::LOGDebug)
    .def("sendCmd",              &stl_intrp_embed_engine_python_py::sendCmd)
    .def("getCmdResp",           &stl_intrp_embed_engine_python_py::getCmdResp)
    .def("recvCmd",              &stl_intrp_embed_engine_python_py::recvCmd)
    .def("hasRecvCmd",           &stl_intrp_embed_engine_python_py::hasRecvCmd)
    .def("recvFeedback",         &stl_intrp_embed_engine_python_py::recvFeedback)
    .def("hasRecvFeedback",      &stl_intrp_embed_engine_python_py::hasRecvFeedback)
    .def("release",              &stl_intrp_embed_engine_python_py::release)
      .def("delay",                &stl_intrp_embed_engine_python_py::delay)
    .add_property("quiesced",           &stl_intrp_embed_engine_python_py::getQuiesced,
&stl_intrp_embed_engine_python_py::setQuiesced);
}

void stl_intrp_embed_engine_python_py::LOGInfo(std::string str) {
  if (myCppP) {
    stringstream ss;
    ss << "PY: " << str;
    myCppP->LOGInfo(ss.str());
  }
}
void stl_intrp_embed_engine_python_py::LOGWarn(std::string str) {
  if (myCppP) {
    stringstream ss;
    ss << "PY: " << str;
    myCppP->LOGWarn(ss.str());
  }
}
void stl_intrp_embed_engine_python_py::LOGDebug(std::string str) {
  if (myCppP) {
    stringstream ss;
    ss << "PY: " << str;
    myCppP->LOGDebug(ss.str());
  }
}
void stl_intrp_embed_engine_python_py::reportError(std::string str) {
  if (myCppP) {
    stringstream ss;
    ss << "PY: " << str;
    myCppP->reportError(ss.str());
  }
}
```

Appendix D. Python driver code

csr_write:

```
    # instantiate a cmd
     csrWriteCmd  = stl_intrp_embed_ctrl_py("csr_write")
     csrWriteCmd.rspReq = True
     csrWriteCmd.setKVP("addr", hex(waddr))
     csrWriteCmd.setKVP("data", hex(wdat))
     csrWriteCmd.setKVP("byte_count", str(bc))
     csrWriteCmd.setKVP("csr_name", csr_name)    #str(csr._name))
     if ( idx != -1 ):
         csrWriteCmd.setKVP("inst", str(int(idx)))
     csrWriteCmd.setKVP("func", func_str)

     # send the command to the cpp side
     ie_sendCmd(csrWriteCmd)
     # get the result
     csrWriteResp = ie_getCmdResp(csrWriteCmd.id, True)
```

csr_read:

```
     # instantiate a cmd
     csrReadCmd  = stl_intrp_embed_ctrl_py("csr_read")
     csrReadCmd.rspReq = True
     csrReadCmd.setKVP("addr", hex(raddr))
     csrReadCmd.setKVP("byte_count", str(bc))
     csrReadCmd.setKVP("csr_name", str(csr_name))

     if idx != None and idx != -1:
         csrReadCmd.setKVP("inst", str(int(idx)))

     csrReadCmd.setKVP("func", func_str)

     # send the command to the cpp side
     ie_sendCmd(csrReadCmd)

     # get the result
     csrReadResp = ie_getCmdResp(csrReadCmd.id, True)
```

Appendix E. SV development command processing

```
    // Main task - just pull stuff off the fifo and analyze the coverage sent
    task processCmd(stl_intrp_embed_ctrl_item ctrlIn, output bit processed);

      stl_intrp_embed_ctrl_send_one_seq   ctrl_send_seq;
      string idx;

      // see if base class handles this command
      processed = 0;
      super.processCmd(ctrlIn, processed);

      if (processed) begin
        `ovm_info(get_type_name(), "base class Processed Cmd from IntrpEmbed",
OVM_FULL)

        return;
      end

      `ovm_info(get_type_name(), "Processing Cmd from IntrpEmbed - SandBox", OVM_FULL)

      if (ctrlIn.name == "CSR_READ") begin

        `ovm_info(get_type_name(), {"Got a CSR_READ command : ", ctrlIn.sprint},
OVM_NONE)

        ///////////////
        // Send the response
        ctrl_send_seq                     =
stl_intrp_embed_ctrl_send_one_seq::type_id::create("cmd_response", this);
        ctrl_send_seq.ctrl.id             = ctrlIn.id;
        ctrl_send_seq.ctrl.name           = "command_response";
        ctrl_send_seq.ctrl.kvp["result"]  = "0xF2F342A100003500";

        if (ctrlIn.kvp.first(idx) )
          do begin
            ctrl_send_seq.ctrl.kvp[idx] = ctrlIn.kvp[idx];
          end

          while (ctrlIn.kvp.next(idx));

        `ovm_info(get_type_name(), {"sending response : ", ctrl_send_seq.ctrl.sprint},
OVM_NONE)

        ctrl_send_seq.start(sqr_h);

        // mark this as processed
        processed  = 1;
      end else if (ctrlIn.name == "CSR_WRITE") begin
            `ovm_info(get_type_name(), {"Got a CSR_WRITE command : ", ctrlIn.sprint},
OVM_NONE)
            ///////////////
            // Send the response
            ctrl_send_seq                     =
stl_intrp_embed_ctrl_send_one_seq::type_id::create("cmd_response", this);
            ctrl_send_seq.ctrl.id             = ctrlIn.id;
            ctrl_send_seq.ctrl.name           = "command_response";
            ctrl_send_seq.ctrl.kvp["result"]  = "1";
            if (ctrlIn.kvp.first(idx) )
               do begin
                  ctrl_send_seq.ctrl.kvp[idx] = ctrlIn.kvp[idx];
               end
            while (ctrlIn.kvp.next(idx));

            `ovm_info(get_type_name(), {"sending response : ",
ctrl_send_seq.ctrl.sprint}, OVM_NONE)

            ctrl_send_seq.start(sqr_h);

            // mark this as processed
            processed  = 1;
```

References

[1] E. Singh, D. Lin, C. Barrett, and S. Mitra, “Symbolic Quick Error Detection for Pre-Silicon and Post-Silicon Validation: Frequently Asked Questions,” *IEEE Des. Test*, vol. 33, no. 6, pp. 55–62, Dec. 2016.

[2] Y. Liu, C. Aparicio, and L. Chang, “Shift left – Running Python tests on pre-silicon validation environment,” presented at the Intel Design and Test Technology Conference 2019, 2019, p. 6.

[3] A. Nahir *et al.*, “Bridging pre-silicon verification and post-silicon validation,” in *Design Automation Conference*, 2010, pp. 94–95.

[4] “UVM-ML Open Architecture,” *Accellera Systems Initiative Forums*. [Online]. Available: http://forums.accellera.org/files/file/65-uvm-ml-open-architecture/. [Accessed: 19-Jun-2019].

[5] Cadence Design Systems, Inc. and Advanced Micro Devices, Inc., “UVM-ML Integrator User Guide.” 04-Aug-2014. Available: https://www.academia.edu/11301150/UVM-ML_User_Guide_UVM-ML_Integrator_User_Guide

[6] “1. Extending Python with C or C++ — Python 3.7.4rc1 documentation.” [Online]. Available: https://docs.python.org/3/extending/extending.html. [Accessed: 23-Jun-2019].

[7] “ctypes — A foreign function library for Python — Python 3.7.4rc1 documentation.” [Online]. Available: https://docs.python.org/3/library/ctypes.html#module-ctypes. [Accessed: 23-Jun-2019].

[8] “Boost.Python - 1.69.0.” [Online]. Available: https://www.boost.org/doc/libs/1_69_0/libs/python/doc/html/index.html. [Accessed: 23-Jun-2019].

[9] P. S. Magnusson *et al.*, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[10] Jonathan Newton and Christian Aparicio, “STL Interpreter Embed Tool Reference.” 08-Jun-2018. Intel Documentation.