DatagenDV: Python Constrained Random Test Stimulus Framework

Jonathan George
Microsoft Corporation
jonathan.george@microsoft.com

James Mackenzie
Microsoft Corporation
james.mackenzie@microsoft.com

Abstract

This paper describes DatagenDV, a Python framework for generating C compatible test stimulus using YAML and random constraints. The framework leverages multiple open-source projects by tying them together into base classes which can then be extended by its users. This approach highlights Python's capabilities in creating custom verification frameworks with simple user interfaces. Project timeline was significantly reduced by DatagenDV's ability to leverage existing SystemVerilog constraints from block testbenches.

I. Introduction

Data generation is an essential part of design verification (DV). Without sufficient or valid data, verification projects are put at risk of not completely exercising the design under test (DUT). When tasked with creating robust random stimulus testing for C firmware, we went looking for the best available tools. Python's extensive open-source libraries proved to be easy to learn and powerful. *PyVSC* (Python Verification Stimulus and Coverage), which mimics SystemVerilog's constrained randomization feature set, proved exceptionally useful. In the matter of a few days, a complex command structure with 40 interconnected fields was correctly randomized in Python by converting existing SystemVerilog constraints from block tests. The significant development time saved from this approach was then spent turning this success into a framework called "DatagenDV" which could be used on future projects.

The goal of this paper is to highlight the features of DatagenDV and how it leverages its imported libraries to create an easy-to-use framework. Although DatagenDV is written targeting a flow based on YAML and C, each feature is largely independent so it could be adapted to target other use cases. This paper should be useful to those interested in Python or those looking to generate stimulus outside of SystemVerilog. We hope that DatagenDV will be an example to other DV engineers of Python's framework building capabilities and also give them inspiration in their own Python coding projects. DatagenDV's source code can be found at: https://github.com/microsoft/datagenDV

II. Related Work

According to IEEE, Python has consistently been the top programming language for multiple years [1]. It is often used for verification infrastructure in place of older scripting languages. Python dominates hardware adjacent industries such as machine learning, signal processing, and quantum computing and has strong communities creating powerful open-source libraries. One previous DVCon paper explored using Python's data science libraries to empower their scoreboard to perform complex signal processing [2]. Another submission showed interfacing with Python from a SystemVerilog testbench through DPI-C to allow for complex data analysis as an alternative to MATLAB pre-generated content [3].

There are also several open-source projects aimed at enabling Python as a viable option for verification. *cocotb* (COroutine based COsimulation TestBench) aims to fully replace SystemVerilog for simulator control [4]. *pyuvm* is a Python implementation of the Universal Verification Methodology (UVM) built on top of *cocotb* [5].

Although these prior works are related to the effort of this paper, the main difference is these efforts look to replace SystemVerilog or tie Python into SystemVerilog testbenches. DatagenDV instead brings the ideas and strengths of object-oriented stimulus randomization to an embedded C test environment using Python.

PyVSC is an open-source effort to create constrained randomization and coverage similar to SystemVerilog. DatagenDV uses *PyVSC* as its randomization engine [6]. The common alternative to the randomization piece is to leverage SystemVerilog's randomization and dump the results to then be parsed and processed by another script before then being passed onto the C environment. Although this method may be better for some situations, we found that keeping the entire pre-processing flow in Python was simpler to implement.

III. Approach

DatagenDV is an object oriented data generation framework designed as a pre-processing step for C tests using YAML specified test cases. As a Python-based framework, DatagenDV heavily leverages Python's robust and open-source libraries. The primary libraries it uses to accomplish its goals are:

- *dataclasses* [7] to provide familiar SystemVerilog-style class declaration with typing.
- *ruamel.yaml* [8] (a PyYAML fork) to provide YAML parsing which loads/dumps YAML to/from Python classes.
- *PyVSC* [6], an open-source library which enables test writers to write SystemVerilog-style constraints.
- *ctypes* [9] to provide C-compatible types and generate binary data packed for C structs.

DatagenDV was originally designed as an augmentation for an existing flow where C firmware tests had parameters specified in YAML. These C test cases were on chip firmware and used for both simulation/emulation verification and post silicon validation. When a test was run, the YAML parameters were converted into a C file with a struct created with values matching the YAML. This conversion was done by a script which existed prior to DatagenDV. The user had to provide a header file containing the struct definition. These files were then compiled with the rest of the C firmware. See Figure 1's dotted box outlining the flow prior to DatagenDV's introduction. DatagenDV streamlines this flow by creating a pre-processing step where test writers can generate more complex data for the C firmware to consume (e.g., raw data, instructions, memory layout, register values).

When generating data for design verification in C firmware tests there are many drawbacks: random constraint specification in C is not as fully-featured as SystemVerilog; generation is performed using the DUT itself, which introduces ambiguity about the validity of the data; and it is difficult to validate that the generated data matches the expectations of the test. In the case of pre-silicon verification, it is also beneficial to avoid extra cycles in the simulator to generate stimulus. Moving this data generation step from C to Python eliminates these drawbacks entirely.

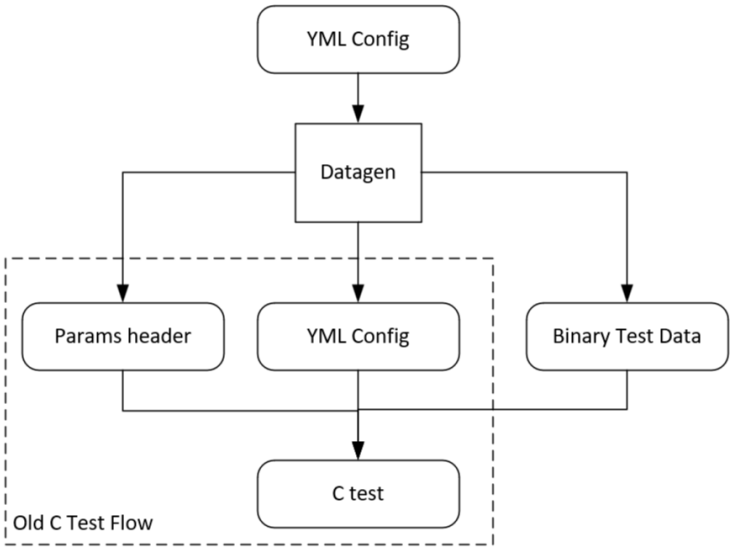

Fig. 1. DatagenDV data flow.

The overarching goal of DatagenDV was to allow test writers to focus on the data they want to create and worry less about the underlying workflow. DatagenDV handles loading and dumping of test configuration, supports randomization, and generates the supporting test definitions. Test writers specify their test configuration schema, the data they need to generate based on it, and the constraints for that data. They do this by creating classes extending the `YAMLParamsBase` class which DatagenDV provides.

A. Building a Python Framework

Part of why Python has become so popular is because of its ability to create complex frameworks with easy user interfaces. There are a few advanced Python concepts which are very helpful when creating frameworks. The first is its reflection capabilities. This is important in a dynamically typed language since functions like `type()` or `callable()` help identify what kind of data is passed into a function in order to appropriately process it. For even more dynamic and flexible functionality, the `vars()` function allows code to access a dictionary of the variables in a class.

The second advanced Python feature which is heavily used in framework building is decorators. Decorators tag a class or function to modify or extend its behavior in some way. To those familiar with Specman e, there are many uses for decorators which give Python aspect-oriented functionality to modify and adjust classes after they have been declared. Similar to UVM macros in some aspects, decorators tend to be more elegant and powerful. The *dataclasses* built-in library in Python is one great example of how decorators extend the language in interesting ways.

```python
import datagenDV as dg
from dataclasses import dataclass

@dataclass
class DatagenParams(dg.YAMLParamsBase):
    #dg.field(<val>) is equivalent to dataclasses.field(init=True, default=<val>)
    FRAME_COUNT  : int   = dg.field(5)
    FRAME_WIDTH  : int   = dg.field(640)
    FRAME_HEIGHT : int   = dg.field(480)
    ITERATIONS   : int   = dg.field(1)

    #These parameters do not need to be dumped, so  dir='in' is used
    #Use lambda to pass a factory function and avoid a mutable default values
    MEM_PADDING  : list  = dg.field(lambda:["RAND"], dir='in')
    MEM_REGIONS  : list  = dg.field(lambda:["RAND"], dir='in')

    #FRAMES is generated by DatagenDV so use dir='out'
    FRAMES : list = dg.field(lambda:[], dir='out')
```

Listing 1. DatagenDV `YAMLParamsBase` implementation specifying YAML fields.

B. Familiar Class Structures with Dataclasses

In most object-oriented languages like SystemVerilog, class variables are declared up front and with static typing. Transitioning to Python's dynamic typing and "declare as you go" style can be an abrupt change of pace. This leads to engineers often avoiding Python classes and overusing Python dictionaries instead, leading to code that doesn't scale and is harder to maintain.

Python's *dataclasses* library provides support for a more SystemVerilog-style class definition. *dataclasses* was added to Python's core libraries in version 3.7. Being a newer and optional feature, it is often overlooked by trainings and tutorials. Because *dataclasses* is a very flexible, extendable library, DatagenDV uses it as a foundation to connect all its other libraries in a straightforward interface. Verification engineers would be wise to consider *dataclasses* when starting their own Python projects.

Similar to `uvm_object_utils` macros in UVM, the `@dataclass` decorator processes the declared class and auto-generates a number of useful features. For each variable of a class, the user provides the name, a type hint and optionally a default constructor value. `@dataclass` generates the constructor (`__init__()`), print, comparison, and sorting functions. It also keeps track of the declared typing of each variable. Although Python does not enforce type hints by default, DatagenDV's `YAMLParamsBase` checks loaded data against type hints during `__post_init__()`. The type hints also provide essential information for DatagenDV to convert Python classes into C structs.

Default values for class variables are specified using `Dataclasses.field()`. DatagenDV provides a wrapper for the `field()` function, extending it to specify additional attributes. For example, specifying whether or not a field should be excluded when loading from or dumping back to YAML. These wrappers also safeguard users against the "common gotcha" of using a mutable default value, such as a list [10]. As seen in Listing 1, *dataclasses* along with DatagenDV's wrappers provides a simple interface for YAML data.

C. Easy YAML Class Loading and Dumping

YAML is a commonly used markup language for verification tools. It has a strong focus on readability and pairs well with Python due to their shared syntax: scopes based on spacing, lists, and dictionaries. This makes YAML a good choice for specifying test cases at a high level. One common issue seen when using YAML for many different scripts is that the schema often goes undocumented. By encapsulating a script's expected YAML inputs and outputs in a class, users of the script have a clear location to lookup correct usage. DatagenDV also protects the user from typos in their YAML field names and ensures typing matches.

The `DatagenBase` class abstracts the details of loading and dumping of YAML away from the test writer. To parse YAML, DatagenDV uses a fork of the *PyYAML* library called *ruamel.yaml*. We chose *ruamel.yaml* over *PyYAML* due to its feature of preserving YAML comments and formatting which is useful for easy 'diff'ing between original and modified YAML. DatagenDV's `YAMLParamsBase` class provides generic implementations of both `to_yaml()` and `from_yaml()`. These functions are specified by *ruamel.yaml* and are called as needed to load and dump the class to YAML. `from_yaml()` relies on field metadata set by DatagenDV's `field()` function to avoid dumping fields which were only meant for use by the Python framework. This is done by using the `field()` function with `dir='in'`. DatagenDV implements this by leveraging the metadata dictionary parameter of the *dataclasses* `field()` function. By specifying the "`!`" class tag and registering the class with the YAML parser, `from_yaml()` ensures that both `__init__()` and `__post_init__()` are called. If a field should be

```yaml
datagen_param: !DatagenParams
  FRAME_COUNT: 5
  FRAME_WIDTH: 0
  FRAME_HEIGHT: 0
  MEM_PADDING: [2,4,6,8]
  MEM_REGIONS:
  - DRAM1
```

Listing 2. YAML input example to match `YAMLParamsBase` class above.

excluded from the constructor, then the field should specify `dir='out'`. The default value for this parameter, `dir='in_out'`, can be used for values which may be specified as an input but are also passed through to the test. Note that *ruamel.yaml* does not call constructors by default. Calling the constructor ensures the default values from the *dataclasses* field definitions are applied.

D. Constrained Randomization with PyVSC

Now for the exciting part: constrained randomization in Python! *PyVSC* is an open-source Python library which mimics the features of SystemVerilog constraint randomization. Powered by the Boolector SMT solver [11], *PyVSC* can solve complex multi-variable constraints with randomization. For a full account of *PyVSC* features, we recommend reading its online documentation [12]. This paper instead focuses on how DatagenDV integrates *PyVSC* into its framework.

PyVSC relies on Python introspection to find variables using its special types (`rand_int8_t`, `rand_enum_t`, etc.). These must be declared in a class's `__init__()` function. Since the *dataclasses* library generates `__init__()`, we must declare these as additional fields in the same manner as the non-random fields. DatagenDV provides another `dataclasses.field()` wrapper function called `rand_field()` to declare these properly. See Listing 3's `rand_field` usage.

PyVSC's special data types are not very easy to work with outside of constraints, so the DatagenDV flow recommends using separate dedicated fields for randomizing and then casting the data back to standard types. DatagenDV provides a class decorator, `@rand_dataclass`, to modify the user's class to do this automatically. This decorator also applies `@dataclass` and pyvsc's `@randobj` decorator.

In order to use the same stimulus script for a variety of tests without changing the Python code, DatagenDV provides a way to overwrite randomized fields to fixed values from YAML. By using a prefix of `rand` to denote *PyVSC* constrained fields, `@rand_dataclass` matches each constrained field with a non-`rand` standard Python typed field. `@rand_dataclass` extends the class's `post_randomize()` function to copy the value back into the non-constrained field. If a non-`rand` field's value is not `None` when `randomize()` is called, then `@rand_dataclass` will create an overriding constraint tying the field to the provided value. This allows the user to override values from YAML and control the randomization of the generated YAML. Listing 4 shows this done for the FrameData class. In this example, we are using the constructor directly to simplify it. As discussed previously in section III-C, DatagenDV calls the constructor when loading YAML into specified classes.

```python
import datagenDV as dg
import vsc

@dg.rand_dataclass
class FrameParams(dg.YAMLParamsBase):
    FRAME_SIZE : int = dg.field(None, dir='in')
    FRAME_WIDTH  : int = dg.field(None)
    FRAME_HEIGHT : int = dg.field(None)

    #Type hints are redundant here so they are replaced with the Ellipsis type.
    rand_FRAME_SIZE : ... = dg.rand_field(vsc.rand_bit_t, 64)
    rand_FRAME_WIDTH  : ... = dg.rand_field(vsc.rand_bit_t, 32)
    rand_FRAME_HEIGHT : ... = dg.rand_field(vsc.rand_bit_t, 32)

    @vsc.constraint
    def min_max_range_c(self):
         self.rand_FRAME_SIZE.inside(vsc.rangelist((16, 65536)))
         self.rand_FRAME_WIDTH.inside(vsc.rangelist((4, 256)))
         self.rand_FRAME_HEIGHT.inside(vsc.rangelist((4, 256)))

    @vsc.constraint
    def frame_dimensions_c(self):
        self.rand_FRAME_SIZE == self.rand_FRAME_WIDTH * self.rand_FRAME_HEIGHT
```

Listing 3. *PyVSC* constraints applied to a DatagenDV params class.

```
frame = FrameParams(FRAME_SIZE=2048)
frame.randomize()
print("frame  - ", frame)
frame2 = FrameParams(FRAME_WIDTH=128)
frame2.randomize()
print("frame2 - ", frame2)
----
frame  - FrameParams(FRAME_SIZE=2048, FRAME_WIDTH=16, FRAME_HEIGHT=128)
frame2 - FrameParams(FRAME_SIZE=7936, FRAME_WIDTH=128, FRAME_HEIGHT=62)
```

Listing 4. Overriding constraints through the constructor using DatagenDV's `@rand_YML_override` decorator.

E. PyVSC Randomization Analysis

Although *PyVSC* is very impressive there are a few caveats which are worth mentioning. Neither the time it takes to randomize nor the evenness of the random distribution are as optimized as most SystemVerilog simulator's implementations. We found that for our use case *PyVSC* was more than sufficient, but recommend that anyone looking at *PyVSC* should investigate and vet it against the scale and needs of their project.

Luckily Python makes it very easy to test both performance and random distribution. Listing 5 shows how we can take the `FrameData` class, randomize it many times and then graph the results. The built-in `vars()` function returns a dictionary copy of any object's variables allowing us to handle any *PyVSC* class implementation. *matplotlib* can then be used to create a clear picture of what kind of distribution our randomization is providing. This would be more difficult to do in SystemVerilog, requiring the engineer to either use coverage tools or export the results to be processed by another language (Python in many cases).

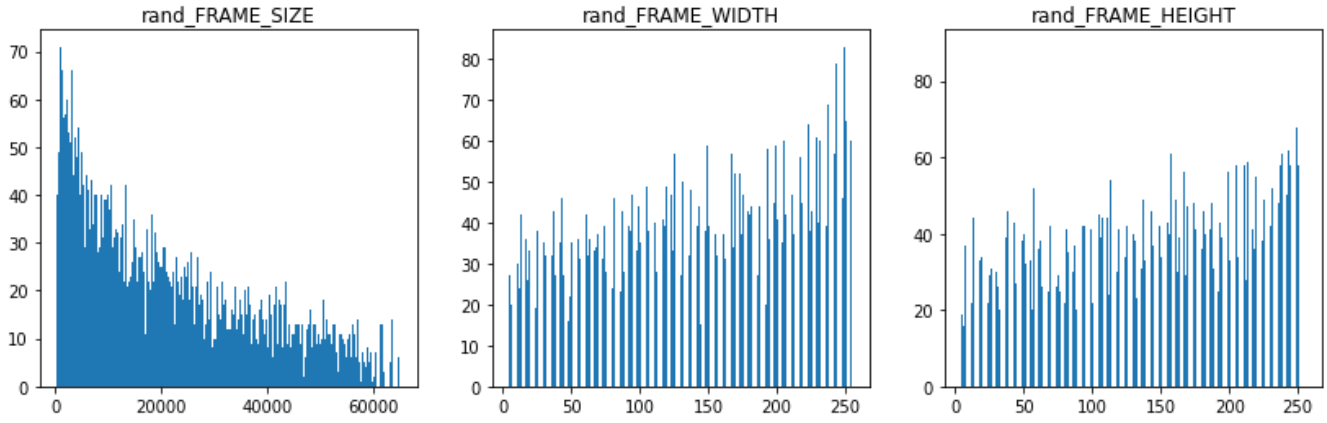

Fig. 2. Histogram of random distribution.

The results from this straightforward example show that *PyVSC*'s current implementation doesn't provide a perfect distribution. However, it doesn't seem to neglect any values and we judged it to perform well enough on our work that it was better than alternatives. The time to randomize this object 10000 took roughly 5 minutes. This is longer than simulator randomization, but shorter than the equivalent randomization on embedded C code. This paper's primary goal isn't to deep dive into *PyVSC*'s performance, but give a quick glance into its current state and how users of DatagenDV can analyze their implementations. It shouldn't be expected that *PyVSC* has stayed the same since this paper was written either. Since starting to investigate *PyVSC*, it has received multiple updates and improvements in response to user bug reports and feature requests.

```
import matplotlib.pyplot as plt

def randomize_and_graph_histograms(rand_class, N=1000):
    class_inst = rand_class()
    var_list = {}
    #Randomize N times, collecting values for graphing
    for i in range(0,N):
        class_inst.randomize()
        class_inst_vars =  vars(class_inst)
        for v in class_inst_vars:
            if v.startswith("rand"):
                var_list.setdefault(v, []).append(class_inst_vars[v].get_val())

    #Create the graphs in rows of 5
    for i,k in enumerate(var_list.keys()):
        if i % 5 == 0:
            fig, axs = plt.subplots(1,5)
        axs[i%5].hist(var_list[k], bins=500)
        axs[i%5].set_title(k)
        if i % 5 == 4:
            fig.set_size_inches(24, 4)
            fig.show()

    if i % 5 != 4:
        fig.set_size_inches(24, 4)
        fig.show()

#Call our generic function with FrameParams and randomize it 10000 times
randomize_and_graph_histograms(FrameParams, N=10000)
```

Listing 5. Creating histograms to visualize random distributions.

F. Bridging the Language Barrier to C

Once data is present inside DatagenDV it needs to be saved out in a compatible format. YAML is often useful since it is human readable and can be passed to further infrastructure scripts. Data stored in a binary format has the benefit of being far more compressed than human readable formats. It can be loaded directly into a C struct without complicated parsing if the bit packing matches. DatagenDV can both generate the binary data and the header by leveraging Python's *ctypes* library. *ctypes* has two main features which are useful to DatagenDV: C compatible types and C memory formatted binary generation. These features seamlessly move data into a C environment from DatagenDV classes.

For more precise typing, DatagenDV users can replace standard Python type hints with *ctypes* types like `c_int32`. DatagenDV can then generate the struct definition for the classes implemented in Python (Listings 6 and 7). To limit redefining common definitions, DatagenDV also supports converting Python constants and enums into C defines and enum typedefs. These features help test writers greatly reduce their maintenance cost and eliminate opportunities for mismatches between the Python and C data structures.

The *ctypes* library also facilitates the creation of binary files from Python structures using the types mentioned above. These binary files are useful when there are external constraints on the size of a test's compiled binaries. An example of generating binary output is shown in Listing 8. These binary files can then be loaded into the C test by using `fopen()` (Listing 9), then using the C header generated by DatagenDV a pointer to the binary data can be created and used inside the C test. It is important to note the memory packing defaults of your compiler and ensure that the *ctypes* library matches otherwise data will not be properly loaded.

```
import datagenDV as dg
import ctypes

@dataclass
class FrameData(dg.YAMLParamsBase, ctypes.Structure):
    width : ctypes.c_uint32
    height : ctypes.c_uint32
    chksum : ctypes.c_uint32
    padding : ctypes.c_uint32
    location : MEM_REGIONS_E

class MEM_REGIONS_E(IntEnum):
    ROM = 1
    RAM = 2
    CD = 3
    FLOPPY = 4
    MAGNET_TAPE = 5
    RAND = 6

//Creates ctype fields array
FrameData.generate_hfields()
with open("frame.h", 'w') as fh:
    dg.ctypes_helper.write_enum_type_defs(fh, enums=[MEM_REGIONS_E])
    dg.ctypes_helper.write_ctype_structs(fh, [Frame], use_hfields=True)
```

Listing 6. Defining a class with *ctypes* and generating a struct definition header file.

```
typedef enum {
  ROM = 1,
  RAM = 2,
  CD = 3,
  FLOPPY = 4,
  MAGNET_TAPE = 5,
  RAND = 6
} MEM_REGIONS_E;

typedef struct FrameData {
    UINT32          width;
    UINT32          height;
    UINT32          chksum;
    UINT32          padding;
    MEM_REGIONS_E   location;
} FrameData;
```

Listing 7. Generated C struct definition based on Listing 6.

```
myFrame = FrameData( width=0xABCD, height=0x1234, chksum=0x9876,
                     padding=0x5432, location=MEM_REGIONS_E.RAND)
ctypes_helper.write_ctype_obj_binary(myFrame, "framedata_output.bin")
```

Listing 8. Python program generating a binary file.

```
#include <stdio.h>
#include <stdlib.h>
#include "frame.h"
int main()
{
    FILE* frame_fh = fopen("framedata_output.bin", "rb");
    FrameData* myFrame = malloc(sizeof(FrameData ));
    fread(myFrame, sizeof(FrameData), 1, frame_fh);
    printf("width 0x%x height 0x%x chksum 0x%x padding 0x%x location 0x%x\n",
      myFrame->width, myFrame->height, myFrame->chksum, myFrame->padding,  myFrame->location);
    fclose(frame_fh);
}
----
width 0xabcd height 0x1234 chksum 0x9876 padding 0x5432 location 0x6
```

Listing 9. C program loading a binary file.

IV. Project Results

Multiple testbenches at Microsoft have leveraged DatagenDV to create more complex test stimulus for their C tests. Test writers have reported that it took them a relatively short ramp up period and that they were much more productive using Python to generate test stimulus.

Prior to DatagenDV being formalized in a generic framework, portions of its solution were used in previous projects for C chip level stimulus generation. In one project, a C struct with interdependent fields had randomization control parameters described in YAML, was randomized in Python, and was written out to a binary object. Leveraging Python's `random` library allowed for significantly more randomization in a shorter wall clock time than if using embedded C in simulation. In this project's case, selecting between 10 possible values for a single enum was 6 orders of magnitude faster in Python than if randomized in C running on a simulated processor.

While it was also easier to write simple constraints in Python's `random` library than in C, describing complex constraints with multiple interdependent values was a consistent issue which limited the scope of testing. The process of implementing and tuning these constraints using Python's `random` library took multiple weeks of effort. If this project had leveraged DatagenDV, reusing block level constraints would have allowed for more complete stimulus generation at the chip level while reducing the amount of effort from weeks to days.

Using this Python-based flow and recognizing the advantages it brought to pre-silicon chip level verification informed the creation of DatagenDV. Recognizing areas of improvement, *PyVSC* was added to allow for its more powerful constraint creation and its similarity with SystemVerilog syntax. The common tasks of parsing YAML and creating C headers and binary files were consolidated into common code for easy reuse. Additionally, switching from dictionaries to classes (using *dataclasses*) provided better abstractions and code reuse.

Taking these lessons learned to the next project, more time was spent upfront designing how Python stimulus generation would be implemented. DatagenDV helped to bridge the gap between UVM based block level testing and C firmware chip level testing. The DUT's stimulus consisted of complex instructions operating on the data in the DUT's memory. DatagenDV was setup to generate both these instructions and the data to be operated on. Creating valid instructions and the data to go with them was very complex. Each instruction consisted of 50+ interconnected fields. Instructions could also be co-dependent through shared memory addresses to create sequences of instructions operating on the same data.

Because *PyVSC*'s syntax mimics SystemVerilog's constraint syntax, UVM block constraints could be directly ported to the chip level. An instruction which contained 60 fields and 300 lines of SystemVerilog constraints was rewritten using *PyVSC* in a day's time, replacing directed test case values. Additional chip level constraints were then layered on top of the block constraints allowing for more complex interconnected randomization. Without leveraging block constraints, it would have taken significant effort and ramp up for chip verification engineers to create a similar level of test stimulus.

This same DatagenDV implementation also supported workflows from software and architecture models. These instructions were simplified and only contained a subset of the full instruction's fields. These tests could be run through DatagenDV by specifying these incomplete instructions in YAML, and then DatagenDV could fill in the blanks using constraint solving. This showed how DatagenDV's approach was flexible, adapting to whatever test parameters were available.

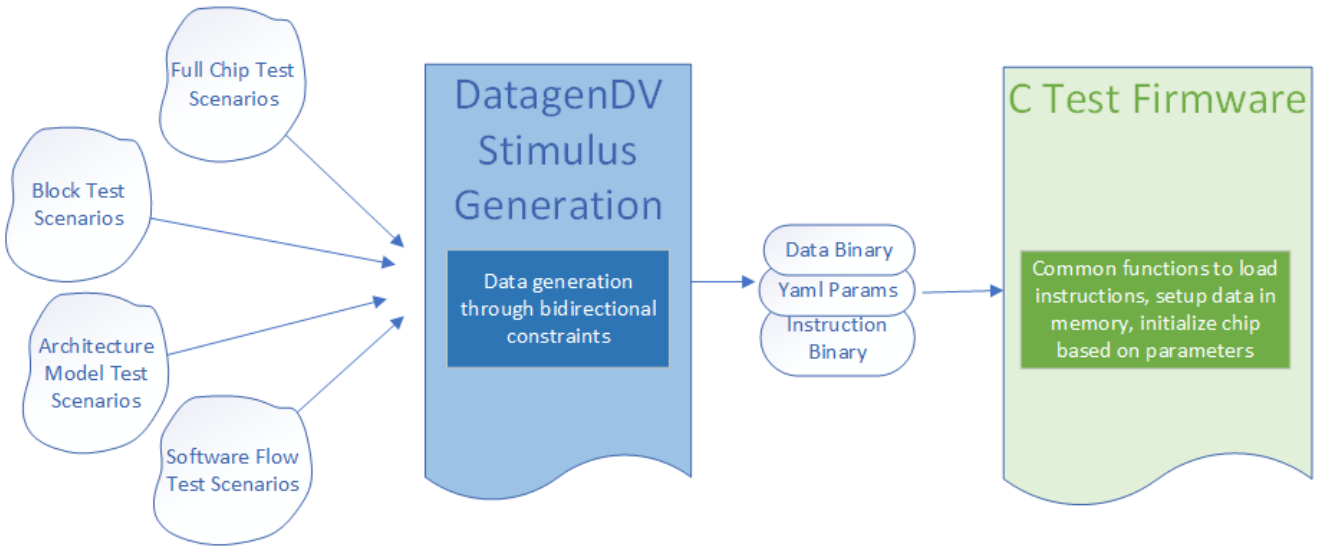

Fig. 3. Multiple test scenarios could be supported through bidirectional constraints.

One challenge initially when using DatagenDV on a large chip team was Python familiarity. Engineers who were already familiar with Python were productive almost immediately. However, those who had only used SystemVerilog were able to use DatagenDV in a reasonable amount of time. They reported that DatagenDV's familiar feel of type declared fields and *PyVSC*'s constraint solving helped them be productive even if they didn't fully understand Python as a language yet. For those without C experience, using Python instead of C was a gentler learning curve due to its modern syntax and easier debug. On this project, we were able to delegate stimulus generation work in Python to those with less C experience and C test firmware work to those with more C experience. This mitigated the impacts of working in so many different languages.

Overall, DatagenDV has shown to be a great benefit for our projects. We have seen it boost productivity and increase coverage capabilities. As we continue to use it, we expect that it will be expanded to support more libraries and use cases.

V. Future Work

DatagenDV is currently still in its infancy and there are many areas which could be improved on. We are already looking at providing more direct support between C and Python. This would involve support for dumping more complex nested data structures from Python to C binary. Currently binary generation is mostly limited to flat structures with basic data types, and *ctypes* doesn't support all packing styles (for example, non-byte aligned). DatagenDV could be enhanced to handle these cases. This would include support and testing for various operating systems and compilers. In the opposite direction, leveraging existing C structs has been looked at with some success at generating *ctypes* Python classes through processing abstract syntax tree generated by Python based C parsing libraries. DatagenDV could also be enhanced to support more direct support between SystemVerilog constraints and *PyVSC*.

Adopters of DatagenDV will likely not have an identical test flow to Microsoft. To support other build flows, DatagenDV will need support for additional configuration languages other than YAML. Supporting languages such as XML or JSON would involve developing base classes to handle seamless parsing and dumping while leveraging DatagenDV's other features. Since Python already has well vetted libraries for most configuration languages, this mainly would involve incorporating it with DatagenDV's framework. This would enable compatibility with other build flows across the verification industry.

Open-source verification tools like *PyVSC* are still new and need additional vetting. We encourage readers to contribute to open source design verification libraries. By creating and improving these tools the industry can become more productive and reduce the in-house reinvention of verification frameworks.

VI. Conclusion

DatagenDV has successfully saved hundreds of hours over several projects. DatagenDV integrates open-source Python libraries to enhance C-based test cases, enabling easy object oriented data generation for more levels of verification. Using YAML as an input to DatagenDV with the *ruamel.yaml* parser allows for easily readable and modifiable test controls. By leveraging the *PyVSC* framework, DatagenDV enables the constrained-random methodologies verification engineers are familiar with and brings them to the C testing environment. Python's *ctypes* library gives DatagenDV the ability to easily generate highly compressed data which can then be loaded into C tests. The *dataclasses* library acts as the foundation by providing a clear interface and extendable metadata. The strength of DatagenDV is combining each of these tools together in an easy to implement framework.

Python's greatest strength is its active open-source culture which enables innovations to build upon each other. We hope that by sharing DatagenDV, other engineers will bring these ideas back into their own projects. As more verification tools and frameworks are open sourced, verification will see more community innovation and higher quality testing options.

References

[1] S. Cass, "Ieee spectrum's top programming languages 2022," Sep 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2022/ieee-spectrums-top-programming-languages-2022

[2] S. Vejlani and A. Chandran, "Challenges in uvm + python random verification environment for digital signal processing datapath design," *DVCon*, 2016.

[3] L. Grover and K. Modi, "Interfacing python with a systemverilog test bench," *DVCon*, 2019.

[4] Contributors, "Welcome to cocotb's documentation!" 2022. [Online]. Available: https://docs.cocotb.org/en/stable/

[5] R. Salemi, *Python for RTL Verification: A complete course in Python, cocotb, and pyuvm*. Independently Published, 2022.

[6] M. Ballance, "pyvsc," Jul 2022. [Online]. Available: https://github.com/fvutils/pyvsc

[7] "Dataclasses - data classes." [Online]. Available: https://docs.python.org/3/library/dataclasses.html

[8] A. v. d. Neut, "Ruamel.yaml," 2019. [Online]. Available: https://yaml.readthedocs.io/en/latest/

[9] "Ctypes - a foreign function library for python." [Online]. Available: https://docs.python.org/3/library/ctypes.html

[10] K. Reitz, "Common gotchas - the hitchhiker's guide to python." [Online]. Available: https://docs.Python-guide.org/writing/gotchas/

[11] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," *Tools and Algorithms for the Construction and Analysis of Systems*, p. 174–177, 2009.

[12] M. Ballance, "Pyvsc documentation," 2022. [Online]. Available: https://pyvsc.readthedocs.io/en/latest/