Sleipnir: Bringing constraints and randomization to software defined data types

Nikhil Soraba
Microsoft Corporation
nikhil.soraba@microsoft.com

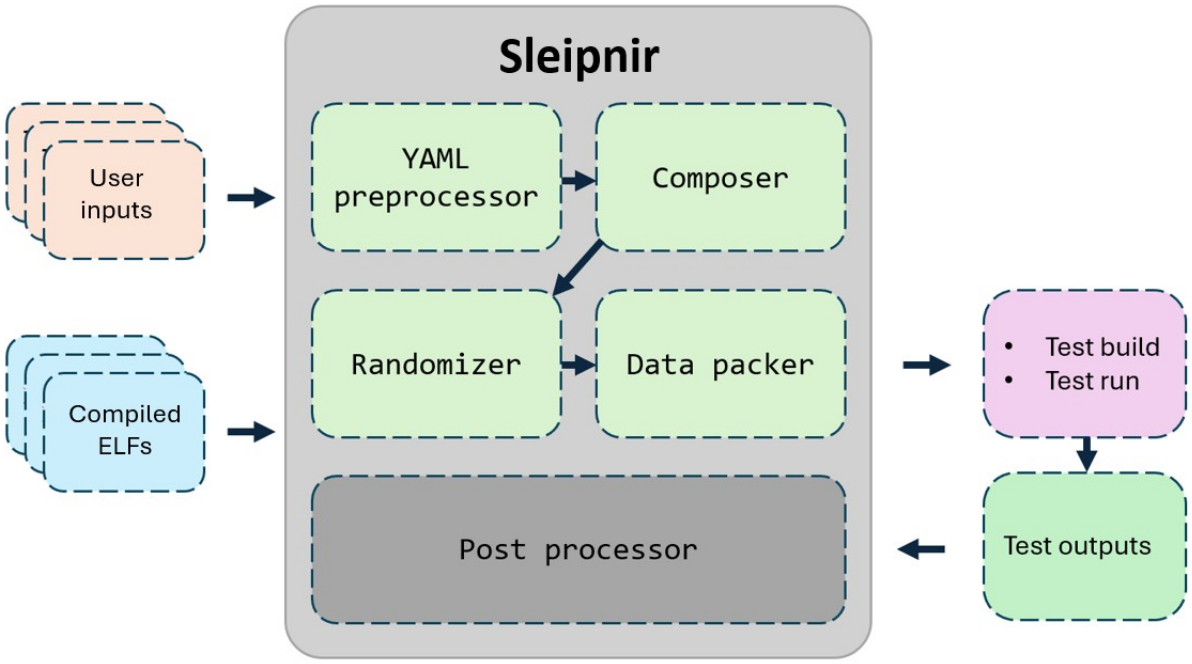
Leon Cao
Microsoft Corporation
leon.cao@microsoft.com

***Abstract*- Low level software-based tests are increasingly used for functional verification of complex SOCs. However, effective randomization of stimulus for these tests continues to be a challenge in the absence of SystemVerilog like randomization features in common programming languages. This paper presents a python library that can randomize complex C/C++ data types based on constraints and distributions, saving time and improving verification quality.**

I. Introduction

Verifying complex SOCs by running software or firmware tests is a proven strategy to identify bugs and gain confidence in the end-to-end working of the design. In this paper, we present an approach to use software defined data types for creating and randomizing stimulus. We demonstrate a method to extract complete information of data types defined in C/C++ and mirror them in Python. Further, we show how PyVSC features can be seamlessly integrated to generate constrained random stimulus. Finally, we look at methods to ingest the generated random stimulus in C/C++ based tests.

Sleipnir's source code can be found at: https://github.com/microsoft/Sleipnir



II. Related Work

[1] introduced PyVSC, a Python library that added native support for constraint and coverage features in Python. The library includes features to provide distribution hints and collect coverage data for generated stimulus. This paper builds on this work by allowing C/C++ data types like booleans, integers, arrays, structs, unions, enums etc. to be represented by equivalent classes for PyVSC.

[2] introduced DatagenDV, a python based constrained random test stimulus framework. The authors present a method for generating C compatible test data from Python scripts. This paper builds on this work by extending the support for stimulus and also allowing C/C++ data types to be automatically imported to python world. This saves time by avoiding repetition of same collateral in two languages. It also avoids expensive debug by allowing for a single source of truth to be maintained and shared by architecture, design, verification and firmware teams.

III. DWARF Extraction

DWARF [3], which stands for "Debugging with Attributed Record Formats", is a widely used debugging standard that offers a rich set of features for data representation and manipulation. It allows for the efficient handling of complex data structures and provides detailed insights into the verification process. DWARF is particularly useful for extracting type information, which can then be used to create stimulus.

In DWARF debugging format, Debugging Information Entries (DIEs) are used to represent various entities in the source code, including structs, unions, and enums. Each DIE consists of a tag and a series of attributes that describe the entity. These DIEs are organized in a tree structure, where each node can have children or siblings, representing the hierarchical nature of the source code.

DWARF Extraction Steps

When compiling with GCC or LLVM, DWARF debugging information is embedded within the ELF (Executable and Linkable Format) file. Here's a high-level overview of the process.

Source Code Compilation: The source code is compiled into object files. During this step, the compiler generates DWARF debugging information when a "-g" option is added to the compile command.

Linking: During the linking phase, the linker combines multiple object files into a single executable or shared library. The DWARF sections from each object file are merged into the corresponding sections in the final ELF file.

Conversion to Python Classes: This ELF file is then loaded in python using `elftools` library. By parsing all the DIE information, equivalent data types are created in Python.

DWARF Die Tag	C/C++ Type	Extracted Attributes		Sleipnir Object
DW_TAG_typedef	typedef	Aliases for other datatypes	-	-
DW_TAG_structure_type	struct	DW_AT_byte_size Child DIEs: 1. DW_TAG_member 2. DW_AT_data_member_location 3. DW_AT_bit_size 4. 4. DW_AT_bit_offset	Struct Size Members Member name Member byte offset Member size Member bit offset	DfStruct
DW_TAG_union_type	union	DW_AT_byte_size die's children: DW_TAG_member	Union Size Members	DfUnion
DW_TAG_enumeration_type	enum	1. DW_TAG_enumerator 2. DW_AT_const_value	Enum name Enum value	DfEnum
DW_TAG_array_type	array	DW_AT_count	Array size	DfArray

IV. PyVSC Integration

PyVSC [4], an open-source library, offers a powerful set of tools for constraint-based randomization in Python. PyVSC maintains its own random state, which ensures consistent results with the same seed. It provides a range of features for managing random stability and solving constraints, making it an invaluable tool for stimulus generation. PyVSC's ability to handle complex constraints and generate high-quality randomization helps us generate stimulus for different use cases.

A. PyVSC Object creation

Continuing from the previous section, each python datatype created after extraction from DWARF is overloaded with PyVSC features to provide support for randomization. This is done by having vsc.randobj decorator.

```
@vsc.randobj
class DfStruct(BfType):
    """Holds a C struct defined in the ELF."""

    def __init__(self: "DfStruct", dwarf_type: dc.Struct, parent: "DfType" = None) -> None:
        """Populate members based on data of the struct parsed from DWARF."""
        super().__init__(parent)

        offsets = collections.OrderedDict()
        for key, value in dwarf_type.members.items():
          ..
          ..
```

B. Adding Constraints

Once the classes are decorated with randobj, we can now add static and dynamic constraints.

```
@vsc.constraint
def base_constr_xyz(self: dp.BfStruct) -> None:
    """Add base constraints for XYZ struct."""

    self.struct_A.field_B <= enm.VALUE_A
    self.struct_B.field_D[3:0].inside(vsc.rangelist(enm.VALUE_B,enm.VALUE_C))
```

Often, constraints are to be selectively applied to only a subset of packets generated. We can use dynamic_constraints to selectively apply them as needed. Here is an example:

```
@vsc.dynamic_constraint
def base_constr_xyz(self: dp.BfStruct) -> None:
    """Add optional constraints for XYZ struct."""

    self.struct_A.field_B <= enm.VALUE_A
```

C/C++ Type	Equivalent Python class	Randomization support	Remarks
typedef	Class pointed to by the typedef	-	-

struct	DfStruct – Decorated with vsc.randobj
	All struct members added as "rand_attr" objects.
	`@vsc.randobj`
	`class DfStruct(DfType):`
	`..`
	`..`
		Constraints allowed on all members of the struct.
		
		Final value determined by combination of all the member values.
			DfStruct

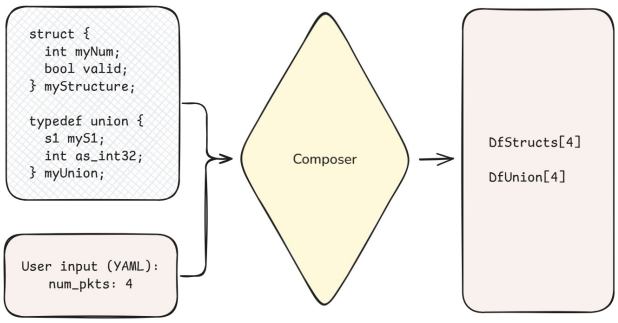
union	DfUnion – Decorated with vsc.randobj
	All members added as "rand_attr" objects. Randomization is enabled (rand_mode) on only one member at a time. Random member picked via precedence rules.
	
	`@vsc.randobj`
	`class DfUnion(DfType):`
	`..`
	`..`
	
	`for member in self.attrs:`
	`if member is not rand_member:`
	`member. set_is_rand(False)`
		Constraints allowed on *only one* member of the union.
		
		Final value determined by whichever member was set last.
		
		When random member is randomized, union's value reflects the randomized value.
			DfUnion

enum	A singleton class that holds all the enums in a dictionary (key: value store). The values can be accessed using object accessor (.) to improve readability. E.g. `abc = enm.MY_ENUM` will assign the value of `MY_ENUM` to variable `abc`.	Static values, no randomization support.	DfEnum

array	Here, we inherit from python's list type to allow indexed accessing and all other list's capabilities for our array class. We also add all the elements of the array as vsc.attr.
	`@vsc.randobj`
	`class DfArray(list, DfType):`
	`..`
	..
		Constraints allowed on all elements of the array.
		Array size is fixed (as in DWARF).
		
		Final value determined by combination of all the member values.
			DfArray

V. COMPOSER

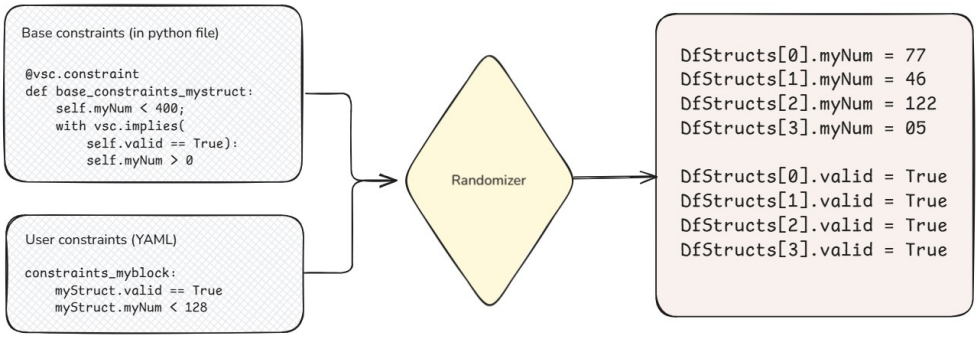
The composer will convert user's intent presented at a high level to a set of collaterals that need to be generated for the test to use at runtime. For example, let us assume a design takes one each of X struct and Y union as input. The composer takes a variable "N" from the user and generates N packets of X struct type and N packets of Y union type. The output is N number of DfStruct type and DfUnion in python.



VI. RANDOMIZER

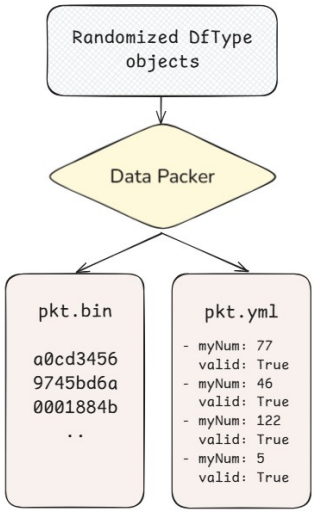
The randomizer is built on pyvsc. It takes in base constraints (static) and additional constraints (dynamic) based on the user inputs. The randomizer can also take constraints for a set of commands (in cases where there are command-command dependencies). The output of the randomizer is consumed by the data packer.

Constraints that are required to generate valid packets are maintained as base constraints in python files. Constraints on top of them that are test to test dependent to exercise a given scenario may be supplied from input YAMLs by the user. Users can also disable randomization on unimportant fields to improve performance and reduce the state-space.



VII. STIMULUS PORTING

Once random stimulus is generated, the values are packed using python's struct module and exported to a file. In verification environment, this file is read using raw C/C++ pointers of the same type to get back values in the software-based tests. For faster debug, helper methods are provided to export the generated stimulus to human readable YAML format. By changing the endianness of the output, the generated stimulus can also be consumed by SystemVerilog functions for replay mechanisms in unit level testbenches.



VIII. Example

In this section, we will take a sample C struct to illustrate the use of Sleipnir for randomization. First, let us look at the setup required and the functions to call to ingest the struct details in python. Next, we will look at ways to provide user input for randomization. Finally, let us look at functions to call to randomize and generate output.

A. Define datatypes in C files

```c
#include <stdio.h>

// Define the Frame struct
struct Frame {
    int width;
    int height;
    int depth;
    int count;
};

int main() {
    // Create an instance of Frame
    struct Frame myFrame;
}
```

B. Generate ELF file

Assuming the above definition is in the file frame.c, we use the following command to generate the ELF.

```
gcc -g -c -o frame.elf frame.c
```

C. Import the struct definitions in Sleipnir's python environment

The following code defines the path to the elf in the filesystem and imports the definitions in the ELF to the python environment.

```
import os
import pathlib

import slp_dwarf_parser as dp

ELF_PATH_FRAME = pathlib.Path().joinpath(
    os.getenv("ELF_PATH"), "frame.elf"
)
"""Path to the ELF file that contains the DWARF information from xCP
cores."""

# Check if Sleipnir's dependencies have been generated first
if not ELF_PATH_FRAME.exists():
    msg = f"{ELF_PATH_FRAME} does not exist."
    raise FileNotFoundError(msg)

dwarfs_frame = dp.parse_dwarf_from_elf(ELF_PATH_FRAME)
dp.enm = dp.BfEnums(dwarfs_frame["enums"])
dp.types = dwarfs_frame["types"]
```

D. User input processing

From Step C, we have C side data types and enums parsed in Python. Let us assume the user input is in a YAML file called input.yml. Let us consider a simple entry for input.yml.

```
sleipnir:
  num_frames: 8
  constraints_frames:
    small_height: frame.height < 128
    odd_width_only: frame.width[0] == 1
```

To ingest this user input in Sleipnir, we call:

```
import slp_preprocessor as pp
suite = pp.process_in_yaml(input_yml)
```

E. Randomization

The following code generates 8 frames and then randomizes according to base constraints and user constraints as specified.

```
import logging
import slp_composer as composer
import frames_randomizer as fr
```

```
    if not (num_frames := user_input.get("num_frames", False)):
        logging.info(
            "'num_frames' for 'frame' not set, skipping generating frames."
        )
        return

    if not (rnd_cfg := user_input.get("constraints_frames", {})):
        logging.info(
            "'rnd_cfg' for 'frame' not set, no user constraints will be
    applied.",
        )

    # Generate frames and randomize them
    frames = composer.gen_frames(num_frames)
    fr.randomize_frames(frames, rnd_cfg)
```

F. Output generation

As a final step, we dump out the generated frames into files. First, we output the frames in packed binary format for consumption by the tests. Second, we also generate a YAML file for human readability and debug.

```
# Generate the bin and yaml files
bin_name_frames, yml_name_frames =
dp.gen_bin_yaml_output_frames(output_path, frames)
```

The generated YAML file will have contents like below.

```
- width: 1
  height: 94
  depth: 42
  num_frames: 8
- width: 55
  height: 39
  depth: 78
  num_frames: 1004
..
..
```

IX. Project Results

Multiple design verification teams at Microsoft have leveraged Sleipnir to generate randomized test stimulus for their C/C++ based tests. Further, teams have successfully used Sleipnir to align on the common format for data types used in the ASIC designs. Sleipnir has been successfully used by Block level and SoC level design verification teams to maintain single source of truth (SSOT) and write complex constraints that dictate generation of valid, randomized samples of such data types.

Before Sleipnir, randomization of packets used two approaches: (a) using “random” module in python to randomize simple integer type datatypes with range constraints and (b) use system Verilog based testbench for complete randomization. Randomization in python allowed quick development with good readability whereas

System Verilog randomization allowed complex constraints at the cost of time intensive setup of testbench environment. Sleipnir strikes the balance in having the readability and maintainability of python coupled with the ability to write complex constraints that span multiple variables. It also aids portability of constraints from block level C tests to SoC level C tests and from project to project. At Microsoft, Sleipnir has been used across two generations of complex SoC projects.

As an example, a complex design block containing over 60 fields that were spread across 13 different structs/unions was ported to Sleipnir. This removed the need to maintain duplicate definitions of these fields in python, helping remove over 2000 lines of code. The constraints for this block that spanned over 5000 lines in SystemVerilog took just 2 days of work for one engineer to write using PyVSC for Sleipnir. Together with similar efforts across 3 other blocks of similar complexity, Sleipnir also enabled generating end-to-end test scenarios where there are inter-block dependencies and constraints. This resulted in us running 10x more randomized scenarios for end-to-end cases as compared to earlier projects, resulting in early detection of critical bugs.

We also noticed significant time savings with Sleipnir. First, the turn around time for any design changes reduced by 60% as any updates to the C source files automatically reflected in the design verification environment, negating the need for manual updates. Second, we saw setup of end-to-end test cases for the next generation of the design took 30% less time due to the ability to update constraints in python along with system Verilog constraints and the ability to port stimulus from block level to SoC level C tests.

X. Limitations and Future Enhancements

There are several areas where this framework can be improved upon. First, it currently lacks the support for parsing preprocessor elements like defines and macros. Since some values that are required for constraints are specified as defines and macros in C, adding this support further improves portability. Second, Sleipnir is written in Python and uses PyVSC as the underlying library for randomization. Due to the speed disadvantages of python and PyVSC, it imposes a cap on the number of constraints and size of randomization that can be performed. Third, special care needs to be taken for definition of structs and unions with members of non-standard widths as DWARF relies on the correct packing and interpretation of these data types by the compiler.

XI. Summary

Sleipnir provides a powerful method to generate constrained random stimulus for software-based testing. It provides a method to seamlessly port type definitions from C/C++ to Python for randomization. Use of Sleipnir results in faster turnaround time for software verification when design or architectural changes occur. It also reduces the need for handwritten code, helping eliminate scope of mistakes and cut verification time. Sleipnir also improves speed of verification by allowing software defined fields to be imported automatically. It improves verification quality by helping cover a larger state space with limited samples. We encourage readers to adopt this open source library for their own design verification needs as well as contribute to it with their enhancements.

References

[1] M. Ballance, “PyVSC: SystemVerilog-Style Constraints, and Coverage in Python” Workshop on Open-Source EDA Technology (WOSET), 2020.

[2] J George, J Mackenzie, “DatagenDV: Python Constrained Random Test Stimulus Framework”, DVCon 2023

[3] *DWARF Debugging Information Format Committee*, “DWARF Debugging Standard,” [Online] Available: DWARF Debugging Information Format (dwarfstd.org)

[4] M. Ballance, “PyVSC Documentation,” 2024. [Online]. Available: https://pyvsc.readthedocs.io/en/latest