plusargs++: Make Plusargs Great … Like They Never Were Before

Bryan Morris, Michael Silveira
Ciena Corporation

***Abstract*-Random constrained verification environments often require a user to provide plusargs to control a simulation's behavior. In *some* verification environments, the number of plusargs explodes to add debug functionality, or to include transient behavior to avoid a bug. In *most* verification environments, the plusargs are not well documented, or are difficult to understand their intent.**

To address these concerns, we use a script that *requires* a user to fully define a plusarg i.e., give it a unique name and type, provide a set of valid choices, and -- most importantly -- add a description. This information is added into a definition file. Our script reads this definition file and *generates* SystemVerilog and C++ code to process the plusarg a user provides on a command line (or in an external `.ini` file). This process includes coercing the input plusarg's string value to its defined type. By generating SystemVerilog code, we can create plusarg types beyond strings: including plusargs that understand integer variants, Enums and real values, file and directory paths, and arrays of Enums, strings and integers. The script also generates documentation for the plusargs defined.

Our build script uses this plusarg++ script and definition file to *validate* the correct use of each plusarg on the command line avoiding the case where a misused plusarg leads to incorrect simulation behavior.

I. INTRODUCTION

This paper defines a common strategy for all plusargs. Addressing how they are created and documented, used within a verification environment, and controlled as an input parameter to control a test variant. It splits the plusarg handling into two parts:

- Requiring the user to fully define the plusarg i.e., name, type, valid values, etc. in an external plusarg definition file (`.ini`) that is processed to generate SV and C++[1] code to process the plusarg from a command line.
- Using the same definition file, we can validate the correct use of each plusarg provided to a simulation. This validation can be used as part of the build process to ensure each plusarg is used correctly – avoiding issues where a plusarg is used incorrectly leading to incorrect behavior of the simulation.

II. SCRIPT FLOW

The script is built with two main features: validation and generation as shown in Figure 1 - plusargs++ Basic Functionality on page 2 :

- **Validation**: reads the plusargs definition file and then validates a user's command-line inputs to build and run a simulation. User input must exactly match the valid choices defined for each plusarg supplied.
- **Generation**: Generates SV and C++ code to provide the run-time parsing of the plusargs from the command line and store them into local 'control-knob' variables for controlling a simulation's behavior.

[1] Our verification environment uses a mixed language of SystemVerilog and C++. The remainder of the paper focusses on the SystemVerilog implementation. The C++ implementation is less interesting as it uses the Boost C++ library for command line processing.

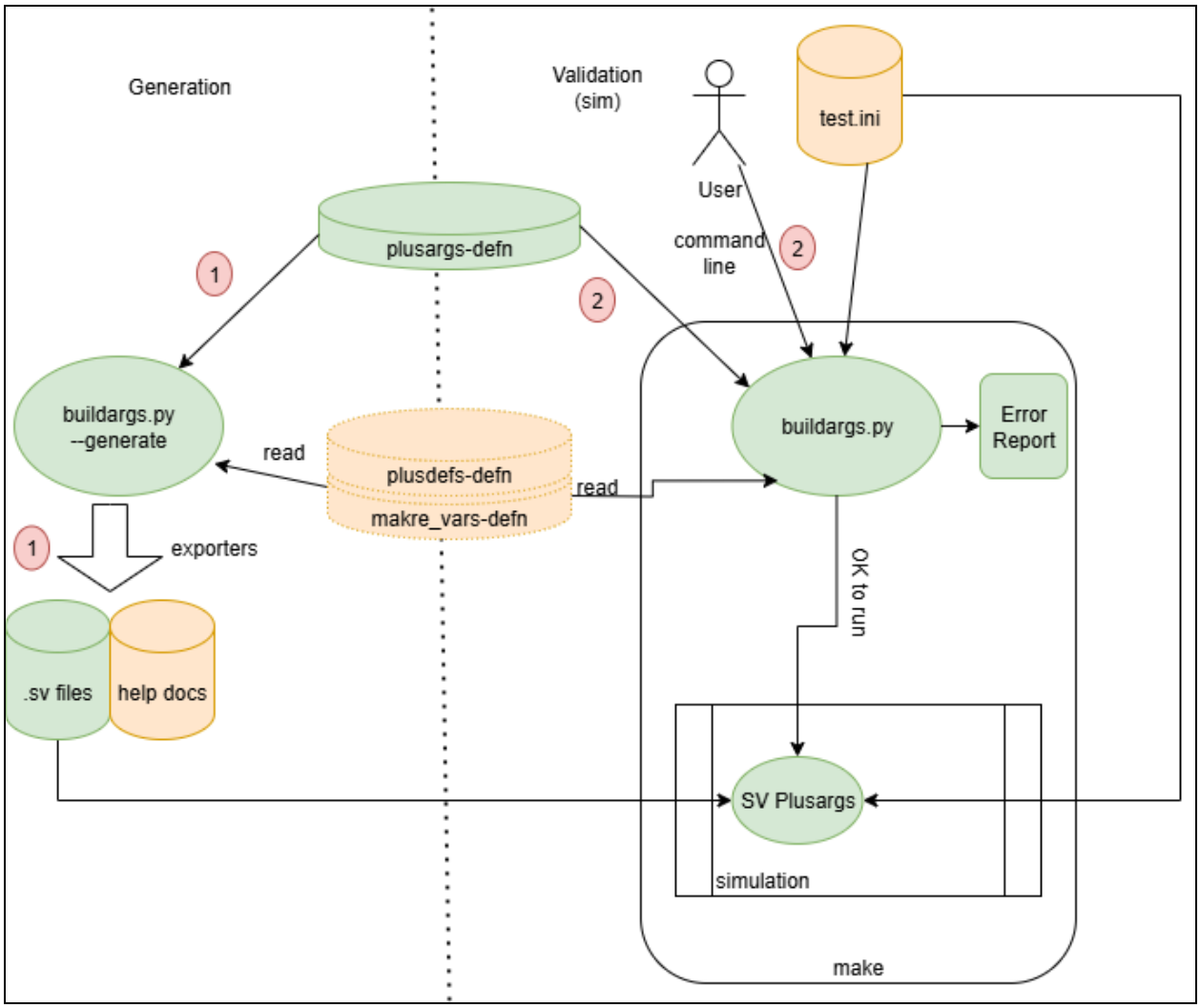

Figure 1 - plusargs++ Basic Functionality

The flow for Generation (left side of Figure 1 - plusargs++ Basic Functionality) is separated into two phases:

- **Import**: Process the definition files and store them in a Python in-memory database
- **Generate**: Template-driven code generation for the SystemVerilog and C++ code: handling type coercion from a string into the appropriate type with result stored into a local variable for use during run-time.

The flow for Validation (right side of Figure 1 - plusargs++ Basic Functionality above) is separated into four phases:

- **Phase 1 - Import**: Same import as generation phase. As shown in the diagram there are three possible input files:
    - `plusargs-defn` – Holds the definitions for all +plusargs to be validated. Where +plusargs are simulation run-time arguments to control behavior. The remainder of this paper describes the design and use-model for this file.
    - `plusdefs-defn` – Holds the definitions for all PLUSDEFs validated. Where PLUSDEFS are compile-time arguments to control compilation. These are not discussed in this paper, but the use-model is like the plusargs definition. PLUSDEFs definitions are limited to validation (not generation).

    - `makevars-defn` – Holds the definitions for all Makefile[2] variables to be validated. These Makefile variables control `make` targets. These are not discussed in this paper, but the use-model is like the plusargs definition. Make variable definitions are limited to validation (not generation).
    -
- **Phase 2 - Transformation**: For convenience, the user can supply a transformation function. This changes the input plusarg string to a new value. Useful when you have a filepath that requires a full directory+file location. If the directory is known, the user supplies the filename portion, and the transformation function prepends the directory path. The Validate phase uses this transformed string.
- **Phase 3 - Validate**: Compares the user input against each supplied plusarg's set of valid choices. Generates an error report (if any compare fails) and exits the build process.
- **Phase 4 - Run-Time**: The generated run-time code reads the supplied plusargs and coerces the string into the correct type for a local variable that the verification environment uses to control simulation behavior.

[2] We currently use Gnu `make` for our build system.

III. SUPPORTED PLUSARG FILE TYPES

Since we generate the SystemVerilog (and C++) processing code for each plusarg, we can extend the type of values a plusarg contains beyond the flag or string values. Table 1 - Supported plusarg types summarizes the currently available types of plusargs we validate and generate.

Table 1 - Supported plusarg types

Plusarg Type	Description	Examples
Integer & integer arrays	Numeric value(s) Provide a (min, max, step) range for validation	`+num_frames=10` `+rate_in_gbps=200`[3] `+per_slice_rates_in_gbps=100,100,200,0`[4]
Real values	Floating point real values	`+pi=3.14159`
String & string arrays	String value(s)	`+chip_name="Titan"`
Enum	Enumerated value(s) – corresponding to SV/C++ enums, single values and arrays of enums	`+single_shot_error=PAYLOAD` `+use_modes=TM1,TM3,TM7`

Convenience Enums: Boolean, enable, toggle	Flag – existence
	boolean = `TRUE | FALSE`
	enable = `ENABLED | DISABLED`
	toggle = `ON | OFF`
		`+JIRA_1234`[5]
		`+stop_after_config=TRUE`
		`+error_injection_en=ENABLED`
		`+debug_pprint=ON`

File and Directory	File and directory locations (include ENV variable processing). (Validation includes file/dir exists)	`+img_file=$GIT_HOME/images/boot.img` `+img_dir=$GIT_HOME/images`

Other types in development are associative arrays and N-tuples of the other types (to capture structures).

[3] **Integer ranges:** The plusarg definition for this range is expressed as (min, max, step) e.g., for `rate_in_gbps` defined as (100,800,100) it checks that value supplied is any value from 100 to 800 (inclusive) in steps of 100 i.e., {100,200,300,..800} e.g., providing 666 causes validation to fail.

[4] **Integer arrays**: Populates an array of integer values (`int unsigned per_slice_rates_in_gpbs[MAX_SLICES]`) for four 'slices' (index value). Slice indices 0 & 1 are set to 100Gbps each, slice index 2 is set to 200Gbps and slice index 3 is set to 0Gbps

[5] **Flag:** This is a typical plusarg where the existence of the flag enables the control.

IV. Python Script Design

The object-oriented design of the Python script relies heavily on Python's `dataclasses` module to simplify the design process. The `dataclasses` module helps create the metadata for each plusarg (see `buildargs_metadata`), bypassing tedious code reuse and copy paste. Figure 2 - Python Script Structure below shows the structure of the script, from the Command Line Interface (CLI) to the base metadata class.

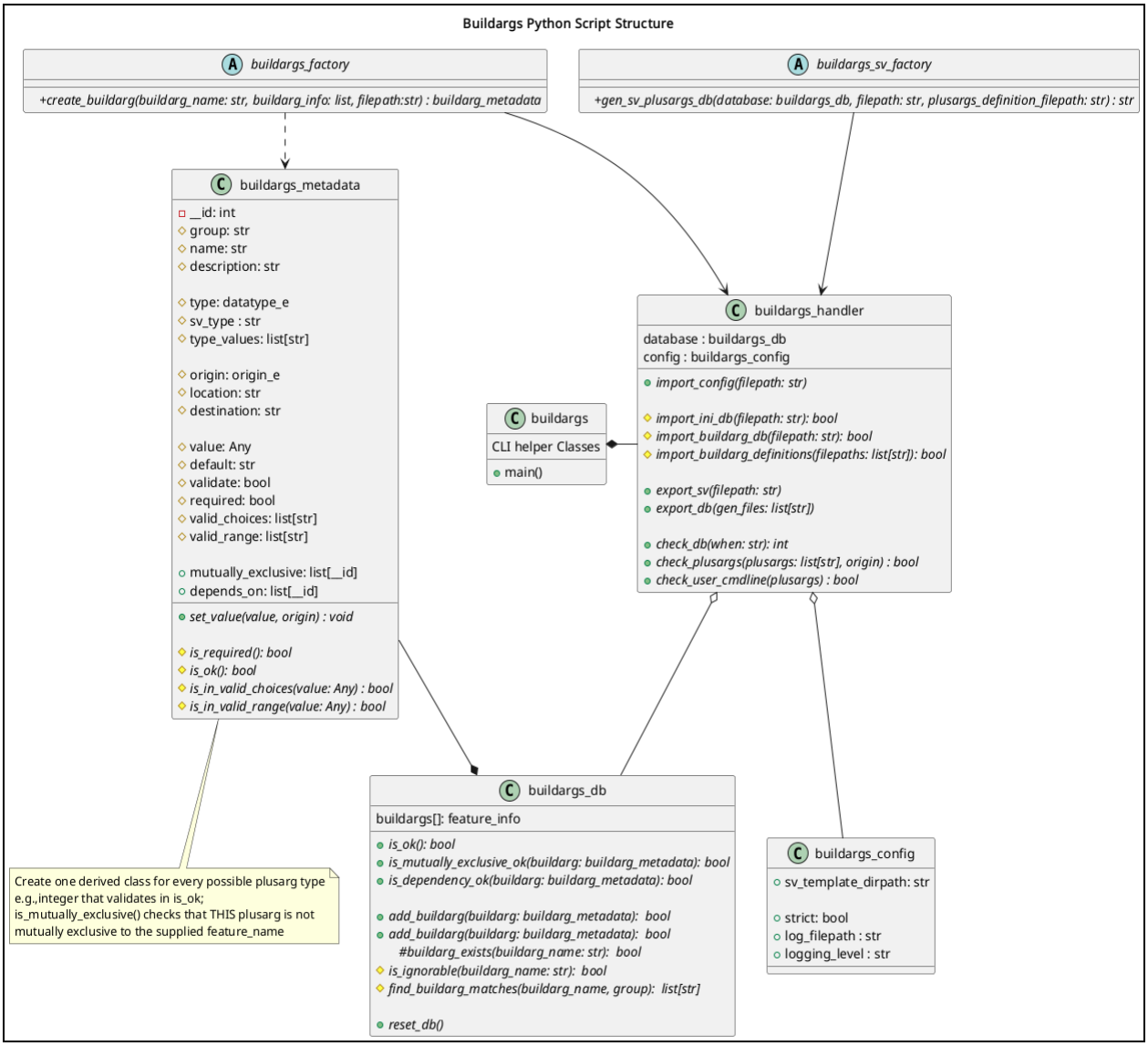

Figure 2 - Python Script Structure

The `buildargs_metadata` class is the base object, containing the fields that any given plusarg might define. Each plusarg type derives from the metadata class, such as `integer_metadata` and `bit_metadata`. Each implements the relevant class methods to provide the correct value verification, casting, and size verification to ensure that the value fits into the SystemVerilog bit size. Performing these checks allows for additional features and checks to be done, such that plusargs like `filepath_metadata` can use Python's `sys` module's `path.exists()` function to ensure the path exists.

The `buildargs_db` class holds a dictionary of the metadata objects keyed by the buildarg's lowercase string name. It provides functions for adding plusargs, checking for the existence of a plusarg by name, removing a plusarg by name, and replacing a plusarg by name with new information. It also offers database functions such as resetting

the database to delete the metadata dictionary, validating each plusarg by calling its `is_ok` function, and checking for mutual exclusivity (i.e., plusargs A and B **must not** be used together) and dependency (i.e., plusargs A and B **must** be used together).

The `buildargs_factory` class creates a `buildargs_metadata` instance, initializes it with the data from the plusargs definition file - given a list of key value pairs. The `buildargs_handler` uses the `buildargs_factory` to create the `buildargs_metadata` objects and add them to the `buildargs_db`.

The `buildargs_config` class provides configuration knobs the `buildargs_handler` uses to control how plusargs are handled. These include, but are not limited to the following:

- enforcing strict validation and generating errors, or non-strict and provide a warning,
- setting the default logging level (can be overridden from the CLI),
- define logging file path, and
- setting the SystemVerilog template directory path.

The `buildargs_sv_factory` uses the `buildargs_db` and the Python String Template module to generate a SystemVerilog class containing all the plusargs as member objects (see Section V. SystemVerilog Design on page 7 for more details). These SytemVerilog objects provide command-line processing, importing from `.ini` files (containing `plusarg=value` settings), and casting of values from command line into the correct SystemVerilog type listed in Table 1 - Supported plusarg types on page 4. The factory also provides an override function to allow the verification environment or testcase to override a plusarg's stored value. A reset function resets a plusarg to its default value.

The `buildargs_handler` class creates the `buildargs_db` instance and a `buildargs_config` object. It has functions to read the config `ini` file, import an `ini` value database for validation. The handler can be used as a Python module or controlled through the CLI. It uses the `buildargs_factory` and `buildargs_sv_factory` to create database entries and export the database to SystemVerilog.

The `buildargs` class is the entry point into the script providing a CLI that wraps the `buildargs_handler` to use it for the build script flow. The `buildargs.py` script uses the CLI exclusively to drive the validation or generation. Our build system (i.e., Makefiles) calls the script and if it returns a non-zero value immediately exits the build before compilation begins.

The `buildargs.py` script is also responsible for the following generation functions:

- declaring and creating all plusarg variables into a single SystemVerilog (or C++) class.
- extracting and type conversion of command-line for each plusarg variable.
- `.ini` file processing and type conversion for each plusarg variable.

It's easier to explain the generation through an example and then work backwards to the class hierarchy.

For the remainder of this section, assume we need three new plusargs in our simulation:

- **`num_of_pkts`** - an integer controlling the number of packets your simulation accepts. e.g., `+num_of_pkts=1000` generates 1000 packets and stops.
- **`mode`** - an enum of type `mode_e` that configures all control register for a simulation's mission mode e.g., `+mode=MODEA` configures the sim for `MODEA` operation.
- **`mem_init_file`** - holds the full filepath to a memory initialization file e.g., `+mem_ini_file~=$GIT_HOME/misc/init/mem_init.patterns.hex`

The definition file `projects_plusrgs.ini` for these three plusargs is shown in code snippet below:

```
[num_of_pkts]
  ;; Example: +num_of_pkts=500
  type = AN_INTEGER
  description = an integer controlling the number of packets your simulation
accepts.

  sv_type = int unsigned
  default_value = 500

  valid_range = [100,10000,1]

[mode]
  ;; Example: +mode=MODEZ
  type = AN_ENUM
  description = an enum of type ~mode_e~ that configures all control register
for a simulation's mission mode.

  sv_type = mode_e
  default_value = MODEZ

[mem_init_file]
  ;; Example:
+mem_init_file=$GIT_HOME/project/misc/mem/init/mem.init.pattern.hex
  type = A_FILEPATH
  description = holds the full filepath to the memory initialization file.

  sv_type = string
  default_value =
```

Figure 3 - Example Plusargs Definition File

Running the generation results in the `project_plusrgs_db.sv` file (Figure 3 - Generated SystemVerilog on page 8) holding the `project_plusargs_db`[6]:

[6] some code removed for brevity.

```
1  class project_plusargs_db extends plusargs_db_base;
2    static project_plusargs_db m_project_plusargs_db;  // singleton instance
3
4    plusarg_INT_UNSIGNED     num_of_pkts;
5    plusarg_ENUM#(mode_e)    mode;
6    plusarg_FILEPATH         mem_init_file;
7
8   `uvm_object_utils(project_plusargs_db)
9
10   function new(input string name="project_plusargs_db");
11     super.new(name);
12     this.initialize_database();
13   endfunction : new
14
15
16   extern virtual function void initialize_database();
17   // Function: set_plusarg
18   //    This function wraps the set functions for each plusarg,
19   extern virtual function void set_plusarg(input string name,
20                                            input string value,
21                                            input plusarg_origin_e origin,
22                                            input string location);
23   // Function: get_plusarg
24   //   Returns the current value of the plusarg name in a string representation.
25   extern virtual function string get_plusarg(input string name);
26   // Function: get_instance
27   //    Returns the singleton instance of this class.
28   extern static function project_plusargs_db get_instance();
29 endclass : project_plusargs_db
30
31 // Function: initialize_database
32 //    This function creates and initializes each plusarg
33 //  NOTE: CREATE_PLUSARGS args: name, type, default_value
34 function void project_plusargs_db::initialize_database();
35   `CREATE_PLUSARG(num_of_pkts    , plusarg_INT_UNSIGNED   , "666")
36   `CREATE_PLUSARG(mode           , plusarg_ENUM#(mode_e)  , "MODEZ")
37   `CREATE_PLUSARG(mem_init_file  , plusarg_FILEPATH       , "")
38
40   this.get_user_plusargs();
41 endfunction : initialize_database
```

Figure 3 - Generated SystemVerilog

The interesting bits from this file are:

Line(s)	Description
1	class derives from `plusargs_db_base` that contains an associative array of plusargs and plusarg processing code.
2	handle to a `static` singleton instance of this database. Ensuring that only ONE instance of the generated `project_plusargs_db` exists.
4-6	the declarations of the three plusargs using the parameterized classes derived from the base `plusarg_ELEMENT` class
11	upon creation the database is populated with the `initialize_database` function.
35-37	`initialize_database()` creates and initializes each plusarg and adds it the associative array of plusargs.

The user includes this file into their compile and creates an instance of the `project_plusargs_db` near the beginning of their `build_phase`. After creation, the `project_plusargs_db` automatically creates and sets the three plusarg variables to their default value. This is overridden if the user sets the plusarg in an `.ini` file (e.g., `num_of_pkts=500`), or on the command-line in one of the supplied `+plusarg` (e.g., `+num_of_pkts=1000`).

The user can access the value either using the `value` member of each plusarg variable (line 5 below), or using the `get()` to return a string representation of the plusarg (line 9 below), as shown in the following code snippet:

```
1  // Get the singleton instance of the plusargs_db
2  my_proj_plusargs_db = project_plusargs_db::get_instance();
3
4  // Configure the mission mode.
5  this.configure_mission_mode(my_project_plusargs.mode.value);
6
7  // Display the mission mode
8  `uvm_info("Configured for mission mode: %s.",
9            my_project_plusargs.mode.get())
```

Figure 4 - Accessing plusarg values

The class hierarchy for the plusarg database is shown in Figure 5 - plusarg_db UML Class Diagram on page 10. The `plusarg_db_base` holds the set of plusarg variables that are parameterized classes derived from `plusarg_ELEMENT`, which in turn derives from the `plusarg_api_base`.

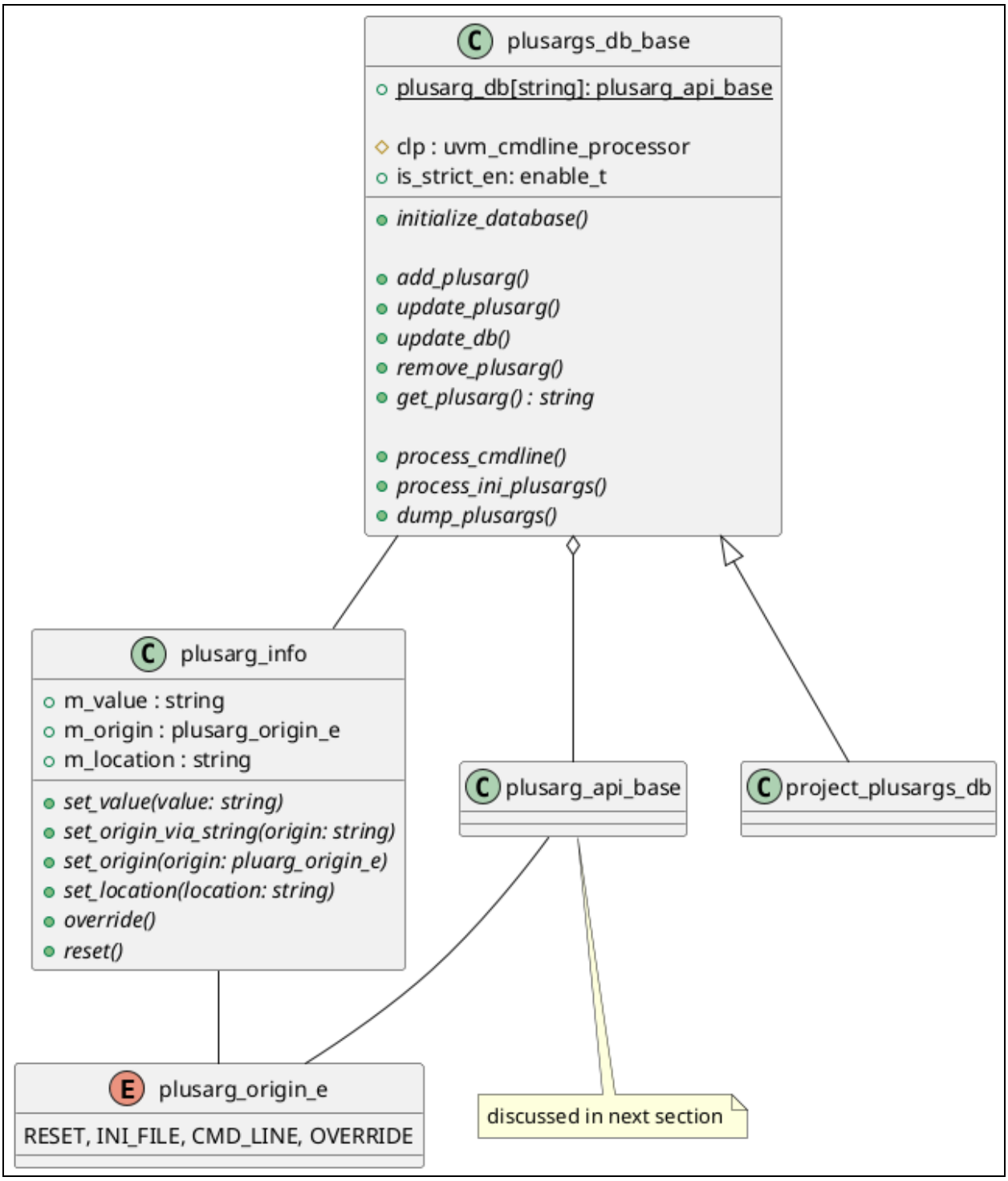

Figure 5 - plusarg_db UML Class Diagram

Working backwards in the class hierarchy, the `plusargs_db_base` provides the following functionality:

- holds an associative array of all `plusarg_api_base` objects i.e., the set of plusarg variables keyed by a unique name.
- a handle to a `uvm_cmdline_processor` for the command-line `+plusarg` processing.
- an `.ini` file processing function to set a plusarg variable's value from a (`plusarg=value`) pair in the `.ini` file e.g., the following `.ini` file would set values of the corresponding plusargs:

```
num_of_pkts=5000
mode=MODEA
```

Each plusarg variable is a parameterized class. The class hierarchy for the `plusarg_INTEGER` is shown in Figure 6 - plusarg_INTEGER UML Class Hierarchy on page 11.

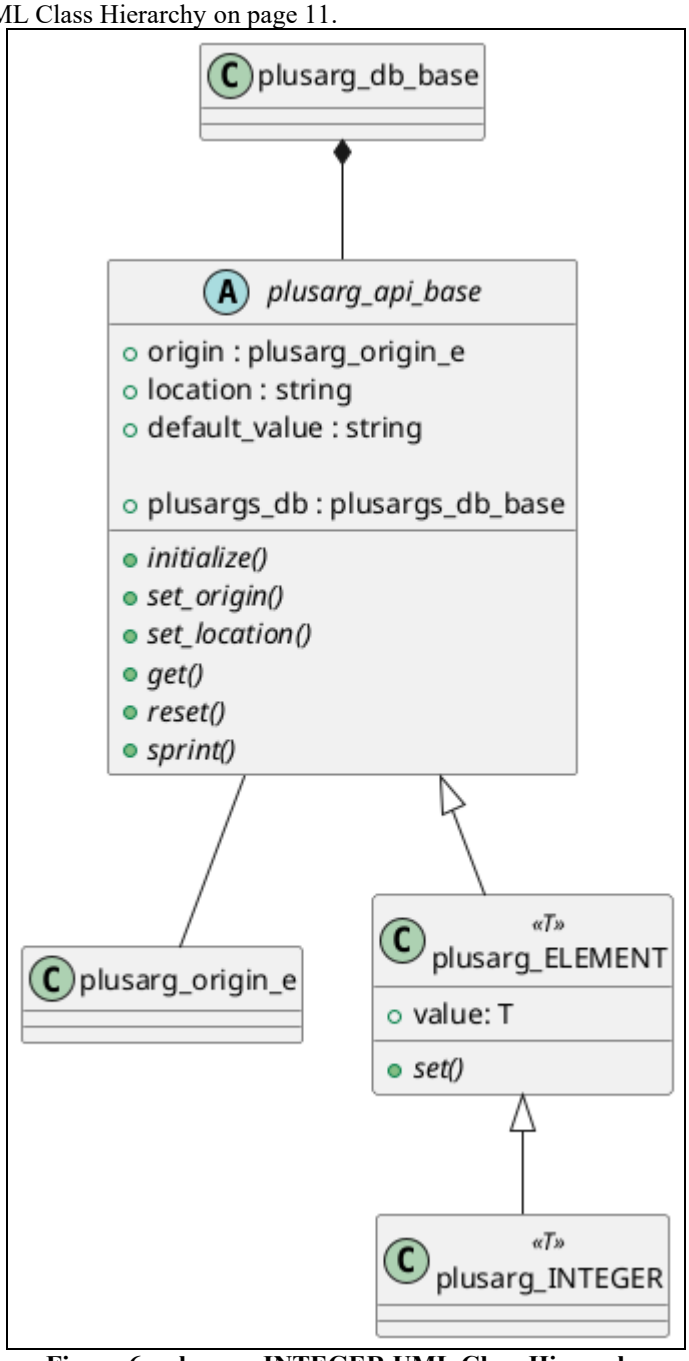

Figure 6 - plusarg_INTEGER UML Class Hierarchy

The `plusarg_INTEGER`'s derives from the parameterized class `plusarg_ELEMENT` holding the correct `integer` storage e.g., `int unsigned`. Which in turn derives from `plusarg_api_base` defining the API to get the plusarg variable's value from the command-line (or optional `.ini` file).

At the top of the plusarg variable hierarchy is the `plusarg_api_base`. This is an abstract base class holding the API for setting and getting a plusarg:

- **`initialize()`** - sets the initial value of the plusarg and adds it to the `plusarg_db`.
- **`set()`** - converts from a string to the proper storage type. Implemented in a derived class.
- **`get()`** - converts from the stored type into an appropriate string representation.
- **`set_origin()`** and **`set_location()`** - provides debugging information about where the plusarg's value was set i.e., default value, `.ini` file, or on the command-line. The location defines the line number in the `.ini` file.

The `plusarg_ELEMENT` parameterized class holds the variable's specified storage type.

Finally, the `plusarg_INTEGER` is a parameterized class deriving from `plusarg_ELEMENT` passing the parameterized type (`integer`) for storage type. It implements the `set()` function to convert a string (i.e., from the command-line or an `.ini` file) into the `integer` type. The conversion converts from any valid Verilog representation into the storage type e.g., `4'hA` `4'b1010`, `'d10`, and `10` convert to a decimal value of `10`.

As shown in Figure 6 - plusarg_INTEGER UML Class Hierarchy on page 11 the SystemVerilog package derives the following scalar plusargs (listed in Table 2 - Available scalar plusargs) from the `plusarg_ELEMENT` and implements the appropriate set/get functionality for each.

Table 2 - Available scalar plusargs

Derivation	Holds...	Example
`plusarg_ENUM`	defined SystemVerilog `enum` variable.	`+mode=MODEA`
`plusarg_BOOLEAN_T`	extends from `plusarg_ENUM` holding the forgotten-from-the-LRM `boolean_t` type i.e., `typedef boolean_t {FALSE=0, TRUE=1};`	`+debug=TRUE`
`plusarg_STRING`	a `string` variable	`+user="Bryan"`
`plusarg_REAL`	a `real` variable	`+clk_freq_ghz=211.123`
`plusarg_INTEGER`	SystemVerilog `integer` type	`+num_of_pkts=100`
`plusarg_FILEPATH`	holds the string defining the filepath. When created it opens the file into the `m_fd` member.	`+mem_init=./allzeroes.hex`
`plusarg_DIRPATH`	holds the string defining the directory path.	`+root_dir=$GIT_HOME`

The aggregate types in Table 3 - Available aggregate plusargs allow us to create plusargs that are comma-separated lists:

Table 3 - Available aggregate plusargs

Derivation	Holds...	Example
`plusarg_INTEGER_QUEUE`	queue of integer values	`+rate_per_ch=800,400,200`
`plusarg_STRING_QUEUE`	queue of `string` values	`+seq_names="seq1,seq4,seq6"`
`plusarg_ENUM_QUEUE`	queue of `enum` values e.g., `plusarg_BOOLEAN_T_QUEUE` holds a list of `boolean_t` enums.	`+loopbacks=LPBK_A,LPBK_E`

VI. Future Work

Future work primarily involves adding other plusarg types (*after* we identify a need). New development includes N-tuples (to allow structs/classes to be configured), and associated arrays of existing types from Table 1.

Other features under consideration are:

- Generating SV & C++ enums from the definition files to ensure correct scoping and use.
- Interactive correction: when validation fails, prompt user to correct the bad data and then re-validate.
- Output to sqlite3 or JSON database (storing an association of all plusargs with a testcase).

VII. Conclusions

At a high level, our script provides two key values:

1. Eliminates time wasted for a simulation supplied with invalid plusarg values.

   Ensuring compile and runtime variables passed in from the command line (or `.ini` files) are within predefined limits.

   Another time-saver is the automatic generation of the SystemVerilog or C++ files that provide the type-casting and command line processing – eliminating the need for a verification engineer to convert plusarg from a string returned by `$value$plusargs()` function to the correct type.

2. Enforcing documentation for a plusarg - *increasing* the probability of its correct use and avoiding the plusarg becoming "stale".

This paper outlines the justification for defining, validating and extending plusarg handling. We describe the design of both the Python script and the base functionality of the generated SystemVerilog classes[7] used during simulation run-time to process the supplied plusargs.

[7] The C++ classes are less interesting because C++ libraries (e.g., Boost) provide most of the functionality not present in SystemVerilog/UVM.