Sharing Generic Class Libraries in SystemVerilog Makes Coding Fun Again

Keisuke Shimizu
ClueLogic, LLC
San Jose, California, USA
keisuke@cluelogic.com

***Abstract*—How many times have you coded a watch-dog timer that triggers when an event does not occur within a time limit? Have you ever wondered whether there was a function to display a few more elements, not just one at an error location, when a data mismatch occurred in a scoreboard? Design verification engineers often encounter such issues, which we addressed by developing a generic class library in SystemVerilog. After surveying several verification projects and other programming languages, we created these commonly used classes: a text processing class, fifteen container classes, two strategy classes, six verification-specific classes, and three domain-specific classes. To evaluate the library's benefit, we categorized our verification code into project-specific and fixed-pattern groups. The preliminary results showed that the library reduced the number of lines of the project-specific code by about 5%. The library was made available as open-source code and placed in a social coding site to encourage the verification communities to extend its functionality further.**

Keywords*—*SystemVerilog, generic library, open source software, text processing, data structures, arrays

I. Introduction

SystemVerilog is among the most widely used object-oriented programming languages in the design verification field [1]. Recent developments in modern verification methodology using SystemVerilog have led to viable reuse of verification components, such as a reusable scoreboard in Universal Verification Methodology (UVM) proposed by Sarkar [2]. Although modern verification methodology like UVM provides a high-level verification framework for this reuse, test-bench development often requires the repeated use of low-level functions. Examples include text manipulation, a linked list to model a chain of DMA descriptors, and data randomization with distributions. These functions can be highly reusable, but lack of an openly available library tends to prevent their reuse. It seems that every test-bench defines its own version of `min` and `max` functions, for example.

Generic functions might be rarely shared for several reasons. One is that most people lack the time to develop such a library. Creating robust functions requires thorough verification. Defining consistent and configurable functions is not trivial, and writing API documents is a burden. Another reason is that some functions are relatively easy to develop. Thus, people do not view creating them as something that is tedious even if it is necessary to do so repeatedly. Lastly, even though the library is created, sharing it might be difficult because of technical and legal restrictions.

Nevertheless, we believe the generic library would benefit the verification community immediately for these reasons: (i) users can save time by not having to develop the common functions; and (ii) they can write more readable code by separating implementation from usage. If the library is created properly, additional benefits might accrue. Foremost, users can avoid common mistakes, pitfalls and gotchas. For example, the library can provide a thread-safe operation if concurrency is crucial. Secondly, users can enjoy the latest SystemVerilog features without knowing them if a simulator supports them. For example, if a simulator supports the `$countbits` system function, the library can delegate the bit counting of a packed array to this function instead of iterating over the array. Thirdly, users can choose different algorithms/implementations based on their requirements. Lastly, the library can provide a uniform way to access data. For example, one can use a common iterator to access a collection class regardless of its implementation. Furthermore, a shared library can evolve and be fixed by peer developers.

This paper discusses the development of a generic library in SystemVerilog that provides frequently used classes and functions. This library is methodology independent. It is made available as open source so that the verification communities can extend it. In order to extract common functions, we investigated multiple verification projects from various domains. We also surveyed other programming languages. We found that modern scripting languages, such as Python and Ruby, have a much richer set of string functions [3], [4] than present in SystemVerilog [5]. Though text processing is not the primary use of SystemVerilog, text handling, such as aligning a text, is not uncommon in test-bench development. Such a manipulation might not be difficult to develop, but it often requires tedious coding. As a result of the investigation, we developed five groups of classes: a text processing class, container classes, strategy classes, verification-specific classes, and domain-specific classes. We are currently investigating 489,875 lines of existing SystemVerilog code, and we found that about 5% of code could be replaced with the classes and functions in the common library. The reduction of code helped to reduce the number of potential bugs. The existing library provides only a limited number of functions. Therefore, the ability to extend the library is one key to its usefulness. Thanks to the recent evolution of social coding sites, code collaboration has never been easier. We used GitHub [6] to make the library open to the communities.

Section II describes the overview of the library. Sections III to VII describe each group of the library and highlight some methods of the library. Sections VIII and IX summarize packaging and code sharing. Section X discusses case studies using the library. Another approach to add functionality is also studied.

II. Overview of the Library

The library was classified into five main groups: (i) text processing; (ii) container; (iii) strategy; (iv) verification-specific; and (v) domain-specific. All groups except the text processing were further divided into sub-groups (Figure 1).

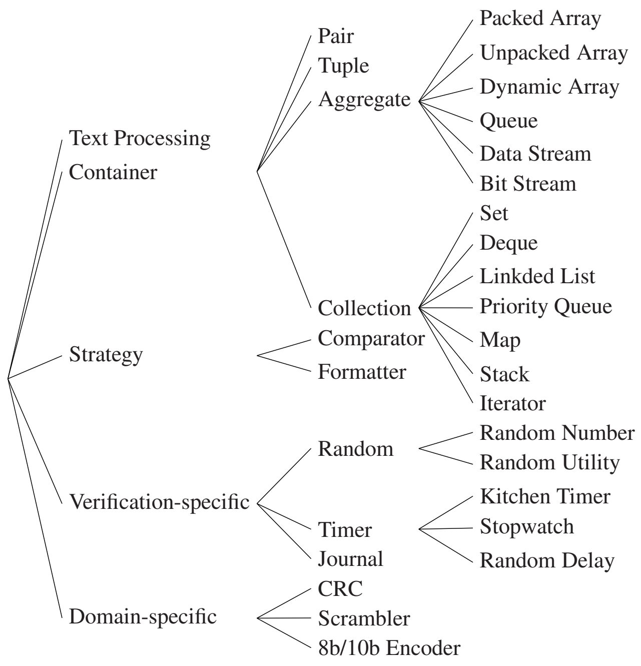

Fig. 1. Groups of the Library

To identify the common functions of the text processing class, the string libraries of C++ [7], Java [8], Python [3], Perl [9], Ruby [4], and JavaScript [10] were surveyed. C++ Standard Library [7] and Java Collections Framework [8] were also surveyed to develop the container classes. In addition, nine verification projects were investigated to extract frequently used functions. The domains of the verification projects included an array processor, an image processor, SoC interconnects, and mobile peripherals.

The library was implemented in SystemVerilog, but some classes allowed the user to choose a foreign language implementation. For example, the text processing class was implemented in two ways. One used native SystemVerilog only, and the other delegated some functions to the Standard C++ Library via Direct Programming Interface (DPI). The former might be selected for easier debugging, while the latter might be selected for better performance.

The library was developed as verification-methodology agnostic. We did not use any methodology-dependent classes in building the library. Many functions were declared as `static` so that they could be used without instantiating an object. All container classes were implemented as parameterized classes, and they could be specialized with a data type. The data type could be an integral data type, such as `bit` and `int`, or a user-defined data type, such as `class`. Unlike Java, primitive wrapper classes were not created intentionally.[1] This allowed us to use a container class for an integral data type without creating a wrapper object.

We focused on low-level classes and functions to lay the foundation for higher-level classes and functions. For example, building blocks of a scoreboard such as a priority queue and a data-stream class were provided, but not the scoreboard itself. There were about 360 functions in the library. In the following sections, we highlighted only the classes and functions that were interesting from a verification point of view. For a complete list of the classes and functions, see our online document [11].

III. Text Processing Class

Although text processing is not the primary focus of verification, we often need it. The number of functions that the string library of other programming languages support varies (Table I). Note that string operators such as "`!=`" were excluded from the number of functions. Also note that the overloaded functions were counted as one.

TABLE I. Number of String Functions

Language	Type/Class/Object	Number of Functions
SystemVerilog	`string`	18
SystemC/C++	`std::string`	25
Java	`java.lang.String`	38
Python	`str`	44
Perl	`string`	22
Ruby	`String`	82
JavaScript	`String`	19

About half of the string functions of SystemVerilog and C++ are numeric converters such as `atoi`. We created forty-seven functions, which were divided into three categories (Table II).

TABLE II. Text Processing Functions

Returns `string`:		
`capitalize`	`join_str`	`slice`
`center`	`lc_first`	`slice_len`
`change`	`ljust`	`strip`
`chomp`	`lstrip`	`swap_case`
`colorize`	`replace`	`title_case`
`contains_str`	`reverse`	`trim`
`delete`	`rjust`	`uc_first`
`insert`	`rstrip`	`untabify`
Returns `bit`:		
`contains`	`is_lower`	`is_upper`
`ends_with`	`is_printable`	`only`
`is_alpha`	`is_single_bit_type`	`starts_with`
`is_digit`	`is_space`	
Returns Other:		
`chop`	`index`	`rpartition`
`count`	`partition`	`rsplit`
`find_any`	`rfind_any`	`split`
`hash`	`rindex`	`split_lines`

The top group lists the functions that return a `string`-type object. The middle group lists the functions that return

[1]A wrapper class wraps around the value of an integral data type to make it an object.

a Boolean `bit`. The bottom group lists the functions that return another type of object. Since we output texts much more frequently than we read texts in SystemVerilog, we created more functions that return a `string` (top group). Nevertheless, the users can do Python-like text processing in SystemVerilog by using the other functions if necessary.

As mentioned in the Overview of the Library, the text processing class was implemented in two ways; one in SystemVerilog only, the other with C++. The following subsections show sample implementations of the `index` function, which returns the index of the first occurrence of the specified substring (`sub`) in the given string (`s`).

A. Implementation 1 – Using SystemVerilog Only

The function uses a `for` loop to find a match:

```
static function int text::index( string s,
                                 string sub,
                                 int start_pos = 0,
                                 int end_pos = -1 );
  int slen = s.len();
  int blen = sub.len();

  if ( slen == 0 || blen == 0 ) return -1;
  normalize( s, start_pos, end_pos );
  for ( int i = start_pos; i <= end_pos - blen + 1;
        i++ ) begin
    if ( s.substr( i, i + blen - 1 ) == sub )
      return i;
  end
  return -1;
endfunction: index
```

B. Implementation 2 – Using SystemVerilog and C++

The function delegates the string search to a C++ function called `c_find`:

```
import "DPI-C" function
  int c_find( string, string, int );

static function int text::index( string s,
                                 string sub,
                                 int start_pos = 0,
                                 int end_pos = -1 );
  int i;
  int slen = s.len();
  int blen = sub.len();

  if ( slen == 0 || blen == 0 ) return -1;
  normalize( s, start_pos, end_pos );
  i = c_find( s, sub, start_pos );
  if ( i >= 0 && i + blen - 1 <= end_pos ) return i;
  return -1;
endfunction: index
```

The `c_find` function creates a `std::string` object and calls its `find` method to find a match:

```
extern "C" {
  int c_find( const char* s,
              const char* sub,
              const int start_pos ) {
    std::string ss( s );
    return ss.find( sub, start_pos );
  }
}
```

Although many functions were inspired by the programming languages we surveyed, some functions were newly created with verification in mind. The next subsection shows an example.

C. Colorize

Some terminals support color. The `colorize` function takes advantage of the color capability. One could use this function to highlight error messages in red in a log file. The function uses ANSI escape codes [12] to change the foreground and background color of the specified text. Additionally, the function can change the font to boldface, underline the text, and even make it blink if supported by the terminal. The formal arguments of the function are shown below:

```
static function string
  text::colorize( string s,
                  fg_color_e fg = FG_BLACK,
                  bg_color_e bg = BG_WHITE,
                  bit bold      = 0,
                  bit underline = 0,
                  bit blink     = 0,
                  bit reverse   = 0 );
```

A sample usage follows:

```
$display( text::colorize( "display me in red",
                          FG_RED ) );
```

IV. Container Classes

A container offers a data structure that collects other objects. Four groups of classes were created: (i) pair; (ii) tuple; (iii) aggregates; and (iv) collections.

A. Pair Class

The `pair` is a parameterized class that carries two values, which can be different types. For example, the code below creates an object of the pair type, which has a value of `int` type and a value of `string` type.

```
pair#(int,string) p;
p = new( 123, "a pair of int and string" );
$display( "%d", p.first );  // first  element
$display( "%s", p.second ); // second element
```

The pair class provides the functions shown in Table III.

TABLE III. Functions of Pair and Tuple Classes

Pair Class	Tuple Class
eq	eq
ne	ne
lt	lt
gt	gt
le	le
ge	ge
clone	clone
swap	swap

B. Tuple Class

The tuple extends the concept of pair. It carries more than two values as a single unit. Since SystemVerilog does not support the variable number of class parameters, we created a tuple class that holds up to ten values. The class declaration looks like this:

```
class tuple #( type T1 = int, type T2  = int,
               type T3 = int, type T4  = int,
               type T5 = int, type T6  = int,
               type T7 = int, type T8  = int,
               type T9 = int, type T10 = int );
  // ... body of the class
endclass: tuple
```

The tuple class provides the same functions as the pair class does (Table III). A pair or a tuple can be used as an argument and/or a return value of a function if more than one value needs to be passed as a single unit.

C. Aggregate Classes

The aggregate classes provide utility functions to various aggregate data types. Six classes were created in this group. Tables IV to VI show the list of functions that each of the aggregate classes supports.

TABLE IV. FUNCTIONS OF PACKED ARRAY AND UNPACKED ARRAY

Packed Array Class	Unpacked Array Class
from_unpacked_array	
to_unpacked_array	
from_queue	from_queue
to_queue	to_queue
from_dynamic_array	from_dynamic_array
to_dynamic_array	to_dynamic_array
init	init
reverse	reverse
count_zeros	compare
count_ones	to_string
count_unknowns	
count_hizs	

1) Packed Array Class: The packed_array is a parameterized class that provides utility functions to a packed array. The class can be specialized with a data type and the width of the array. The data type must be the bit data types (bit, logic, reg), enumerated types, or other packed arrays and packed structures. The following sample code shows how to convert a packed array into an unpacked array using one of the functions of the class.

```
bit[7:0] pa; //   packed array
bit ua[7:0]; // unpacked array
ua = packed_array#(bit,8)::to_unpacked_array( pa );
```

2) Unpacked Array Class: The unpacked_array is a parameterized class that provides utility functions to an unpacked array. The class can be specialized with a data type and the size of the array. The data type can be any data type. The following sample code reverses the order of unpacked array elements.

```
my_class ua[8]; // unpacked array of my_class
unpacked_array#(my_class,8)::reverse( ua );
```

TABLE V. FUNCTIONS OF DYNAMIC ARRAY AND QUEUE

Dynamic Array Class	Queue Class
from_unpacked_array	from_unpacked_array
to_unpacked_array	to_unpacked_array
from_queue	from_dynamic_array
to_queue	to_dynamic_array
init	init
reverse	reverse
compare	compare
clone	clone
split	split
merge	merge
concat	concat
extract	extract
append	append
to_string	to_string

TABLE VI. FUNCTIONS OF DATA STREAM AND BIT STREAM (INHERITED FUNCTIONS ARE NOT SHOWN)

Data Stream Class	Bit Stream Class
to_bit_stream	alternate
make_divisible	count_zeros
sequential	count_ones
constant	count_unknowns
random	count_hizs
scramble	
to_string	
to_string_with_en	

3) Dynamic Array Class: The dynamic_array is a parameterized class that provides utility functions to a dynamic array. The class can be specialized with any data type. The following sample code shows how to compare two dynamic arrays using a user-provided comparator.

```
my_class da1[]; // dynamic array 1
my_class da2[]; // dynamic array 2
bit result = dynamic_array#(my_class)::
  compare( da1, da2, .cmp( my_comparator ) );
```

The my_comparator is a strategy class that provides the functions that compare two objects of my_class type. It must provide at least the ne function that returns 1 if two objects are not equal. If no comparator is given, the elements of the dynamic arrays are compared using the default comparator, whose ne function compares two objects using the logical inequality operator ("!=").

4) Queue Class: The queue is a parameterized class that provides utility functions to a queue. The class can be specialized with any data type. This class provides similar functions to the ones the dynamic array class provides (Table V).

5) Data Stream Class: The data_stream is a parameterized class that manages a stream of packed arrays. The data_stream is a subclass of the dynamic_array class (Figure 2).

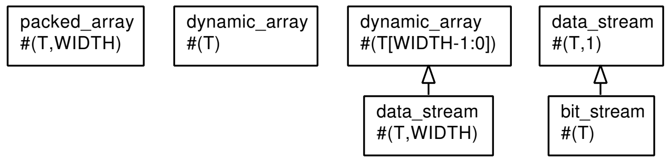

Fig. 2. Specialization of the Array Classes

Similar to the packed_array class, this class can be specialized with a data type and the width of the array. The same restrictions as that of the packed_array class apply in terms of the kind of data type. This class has a function called to_string to display its contents in a variety of ways. For example, the following code displays only four elements from the beginning of the data stream and six elements from the end of the data stream.

```
bit[7:0] ds[]; // data stream
ds = new[16]( '{ 'h00, 'h11, 'h22, 'h33,
                 'h44, 'h55, 'h66, 'h77,
                 'h88, 'h99, 'hAA, 'hBB,
                 'hCC, 'hDD, 'hEE, 'hFF } );
$display( data_stream#(bit,8)::to_string( ds,
  .group( 2 ), .num_head( 4 ), .num_tail( 6 ) ) );
```

The output looks like this:

```
0011 2233 ... AABB CCDD EEFF
```

Note that the group argument specifies the number of elements to group together, the num_head specifies the number of elements at the beginning of the data stream to display, and the num_tail specifies the number of elements at the end of the data stream to display.

A data stream may be accompanied with its data-enables. For example, the code below displays the value of an array element (ds[i]) if a corresponding data-enable (de[i]) is 1. Otherwise, it displays "--".

```
bit[7:0] ds[]; // same data stream as before
bit      de[]; // data enables
de = new[16]( '{ 0, 1, 1, 0,
                 1, 0, 0, 1,
                 1, 1, 1, 1,
                 1, 0, 0, 1 } );
$display(data_stream#(bit,8)::to_string_with_en(
  ds, de, .group( 4 ), .group_separator( "\n" ) );
```

The output looks like this:

```
--1122--
44----77
8899AABB
CC----FF
```

There are more arguments to customize the output (see [11]).

The data_stream class also has a rich set of data-generation functions. For example, the code below generates a dynamic array of 16 elements that have sequential values with a randomized initial value.

```
bit[7:0] ds[];
ds = data_stream#(bit,8)::sequential(
  .length( 16 ), .randomize_init_value( 1 ) );
```

The generated array can be displayed as before (assuming the randomized initial value was 'h75):

```
$display( data_stream#(bit,8)::to_string( ds,
  .group( 1 ), .num_head( 4 ), .num_tail( 0 ) ) );
// 75 76 77 78 ...
```

6) Bit Stream Class: The bit_stream is a parameterized class that manages a bit stream. The bit_stream is a subclass of the data_stream class specialized with a 1-bit width (Figure 2). One can use this class to manage a serial data stream.

D. Collection Classes

The collection classes offer data structures. Seven collection classes and one iterator class were created (Figure 3).

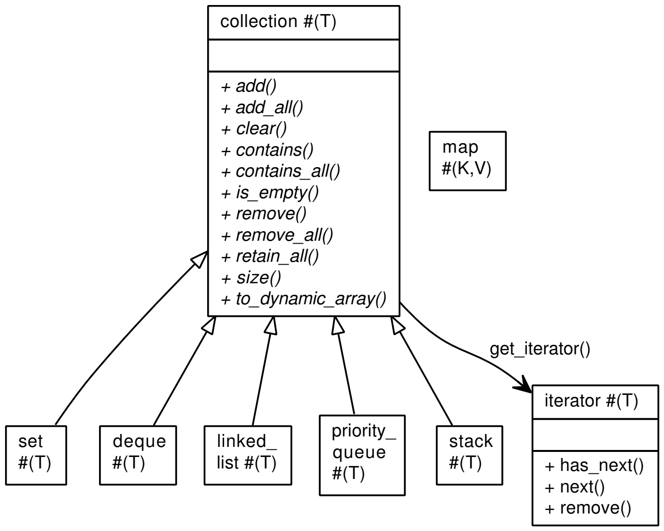

Fig. 3. Collection Classes

The collection class is an abstract base class that defines the common functions that its subclasses should implement. Some collection classes were implemented using a queue and an associative array type in SystemVerilog. The queue and the associative array have associated methods. It is important to know the performance of the methods because selecting an inappropriate method can negatively impact the performance of the collection classes. For example, inserting an element to the front of a queue using the insert function, or deleting an element from the back of a queue using the delete function takes much longer time than other operations.[2] This is because all elements have to be shifted in the queue for these operations. We took these underlying structures into consideration when we created the collection classes described below.

1) Set Class: The set is a collection class that contains no duplicate elements. It is implemented using an associative array.

2) Deque Class: The deque is a double-ended queue class that supports push and pop at both ends. It is implemented using a queue.

3) Linked List Class: The linked_list is a doubly-linked list class. It is implemented using a newly created link element type. The class provides better performance for inserting and deleting elements than the deque class does.

[2] q.insert(0,e) is equivalent to q.push_front(e), but the former takes a much longer time.

4) Priority Queue Class: The `priority_queue` class orders its elements according to their priority. It is implemented using a newly created priority heap.

5) Map Class: The `map` class maps keys to values. It is implemented using an associative array.

6) Stack Class: The `stack` class implements a Last-In-First-Out (LIFO) structure. It is implemented using a queue.

7) Iterator Class: The `iterator` class provides uniform access methods to the collection classes regardless of their implementations. The following code shows an example to iterate over each element within a set.

```
set#(int) int_set;
iterator#(int) it = int_set.get_iterator();
while ( it.has_next() )
  $display( it.next() );
```

V. Strategy Classes

The strategy classes provide a family of algorithms used by other classes. Two groups of classes were created: (i) comparators; and (ii) formatters.

A. Comparators

The comparators are used to compare two objects. They are mainly used by the container classes. A new comparator should be created if two objects need to be deep-compared. For example, the `eq` function of the default comparator uses the logical equality operator (“==”) for the object comparison:

```
class comparator #( type T = int );
  virtual function bit eq( T x, T y );
    return x == y;
  endfunction: eq
  // ... other functions
endclass: comparator
```

The default comparator is fine for comparing two objects of an integral data type, such as `int`. However, if the type is the `pair`, you might want to compare the elements in the pairs, not the object handles. The `pair_comparator` is one of the comparators the library offers. This comparator compares the elements in the pairs to determine the equality:

```
class pair_comparator #( type T = pair )
  extends comparator #(T);
  virtual function bit eq( T x, T y );
    return x.first  == y.first &&
           x.second == y.second;
  endfunction: eq
  // ... other functions
endclass: pair_comparator
```

B. Formatters

The formatters are used to convert an object to a string format. A simulation often uses large integer values, such as simulation time values scaled in picoseconds. The `to_string` function of the `comma_formatter` class can format an integral data value to a string that has commas as thousands separators for better readability. For example, the following code displays “`123,456,789`”.

```
comma_formatter#(longint) com_fmtr
  = comma_formatter#(longint)::get_instance();
$display( com_fmtr.to_string( 123456789 ) );
```

VI. Verification-specific Classes

The verification-specific classes offer utility functions useful for verification. We created three groups of classes: (i) random classes; (ii) timer classes; and (iii) a journal class.

A. Random Classes

1) Random Number Classes: The random number classes provide pre-defined distribution bins for randomizing a number. Classes with 2, 4, 8, 16, and 32 distribution bins are defined. The random number class with four bins is shown below.

```
class random_4_bin_num;
  dist_bin db[4];
  rand int val;

  constraint val_con {
    val dist {
     [db[0].min_val:db[0].max_val] :/ db[0].wt,
     [db[1].min_val:db[1].max_val] :/ db[1].wt,
     [db[2].min_val:db[2].max_val] :/ db[2].wt,
     [db[3].min_val:db[3].max_val] :/ db[3].wt
    };
  }
endclass: random_4_bin_num
```

The `dist_bin` is a `struct` defined as:

```
typedef struct {
  int min_val;
  int max_val;
  byte unsigned wt; // distribution weight
} dist_bin;
```

For example, the following code creates four random distribution bins:

```
random_4_bin_num n = new;

//           min  max  wt
n.db = '{ '{ 100, 200, 1 },   // bin 0
          '{ 300, 400, 2 },   // bin 1
          '{ 500, 600, 3 },   // bin 2
          '{ 700, 800, 4 } }; // bin 3
assert( n.randomize() );
$display( n.val ); // the randomized value
```

The value of variable `n.val` is randomized to between 100 and 200, between 300 and 400, between 500 and 600, or between 700 and 800 with a weighted ratio of 1-2-3-4.

2) Random Utility Class: The `random_util` provides a function to return a value of `bit` type that shows true (`1`) or false (`0`). It is randomized based on the given percentage. In the following example, the conditional expression is randomized to `1` with 70% of probability.

```
if ( random_util::random_bool( 70 ) ) begin
  // ...
```

B. Timer Classes

Table VII lists the functions of the timer classes.

1) Kitchen Timer Class: The `kitchen_timer` class counts a simulation time (not a wall clock time) and triggers an event once the timer expires. The kitchen timer can be used as a watch-dog timer or as an event trigger that fires a later time.

2) Stopwatch Class: The `stopwatch` class also counts a simulation time. Unlike the kitchen timer class, the stopwatch class is mainly used for monitoring the performance of an internal process. The user can measure the end-to-end delay of a transaction, or the duration between two events.

3) Random Delay Class: The `random_delay` class waits for a randomized time within the range specified by the user. The user can specify an event to exit from the waiting state before the randomized delay elapses.

TABLE VII. TIMER FUNCTIONS

Kitchen Timer Class	Stopwatch Class	Random Delay Class
start	start	delay
stop	stop	
pause	pause	
resume	resume	
reset	reset	
is_stopped	is_stopped	
is_running	is_running	
is_paused	is_paused	
set_delay	measure	
add_delay		
set_random_delay		
get_elapsed		
get_remaining		

C. Journal Class

The `journal` class provides a uniform method to write a transaction log. It can save a separate log file for post-processing.

VII. DOMAIN-SPECIFIC CLASSES

The domain-specific classes described below offer utility functions to a specific target domain, but these functions are generic enough to reuse.

A. CRC Class

The `crc` class calculates a cyclic redundancy check value. It provides forty-two different, commonly used CRC functions. One function declaration of the CRC class is shown below:

```
static function bit[15:0]
  crc::crc16_ccitt( bit bitstream[] );
```

A general-purpose CRC function that can use a custom CRC polynomial is also provided.

B. Scrambler Classes

The scrambler is a parameterized class that provides a scramble function to a bit stream. One general-purpose scrambler as well as eighteen commonly used scramblers were created. The function declaration to scramble a bit stream follows:

```
class scrambler #( type T = bit, int DEGREE = 2 );
  typedef T bs_type[]; // bit stream type
  typedef T[DEGREE-1:0] lfsr_type;

  virtual function bs_type scramble(
    bs_type bs, ref lfsr_type lfsr ); // LFSR values
    // ... body of the function
  endfunction: scramble
  // ... other functions
endclass: scrambler
```

The function returns the scrambled bit stream as well as the Linear Feedback Shift Register (LFSR) value, which can be used as the seed value for the next call of this function.

C. 8b/10b Encoding Class

The 8b/10b encoding class is currently being developed. The class provides 8b/10b encoding and decoding functions for serial communications.

VIII. PACKAGING

A. Package

All library files are included by a single file called `cl_pkg.sv`. It packages class definitions into one SystemVerilog package called `cl`. We chose this short name for the package to make it less obtrusive, but one can change the name if it conflicts with other namespaces. A class declared within the package can be accessed using one of the following:

- the class scope resolution operator "`::`"

```
int i = cl::choice#(int)::min( j, k );
```

- an explicit `import` declaration

```
import cl::choice;
int i = choice#(int)::min( j, k );
```

- a wildcard `import` (as usual)

```
import cl::*;
int i = choice#(int)::min( j, k );
```

Many functions were defined as `static` so as to be used without instantiating an object.

B. License

The library is licensed under the MIT / X Window System License [13], which means "this library can be used however you want even in proprietary verification."

IX. SOCIAL CODING

Our library is a first step towards a richer and more complete set of libraries. In the last few years, web-based repository hosting services for software projects have emerged. GitHub [6] is one of the so-called "social coding" service providers. We chose GitHub not only because it is the most popular open-source forge [14], but because it offers more social aspects that encourage developers to contribute. We have identified the following stages in which one becomes involved in social coding:

A. *Downloads*

The initial stage is actually using the library. Downloading is simply a click of a "Download ZIP" button. The whole library package, as well as the test suite used to verify the library, are downloaded.

B. *Comments*

The second stage is to comment on the library. GitHub offers line-level comments to allow the reviewer to make a suggestion to a specific line of code. The other users' contributions can be "watched" similar to the way one can follow Twitter users. Personal notifications to the specific activities can be set as well.

C. *Forks*

The third stage is to modify the library. Forking creates a personal copy of a project. This personal copy can be modified in any way.

D. *Pull Requests*

The fourth stage is to share the modification with the community. Pull requests allow the user to request that his or her contribution be merged.

We anticipate that using GitHub lowers the barriers for contributing at each stage mentioned above. However, there is a complication specific to verification libraries. More details on this will be given in the Difficulties of Open-Source Projects in the next section.

X. Results and Discussion

A total of thirty-eight classes and a total of 362 functions and tasks were created. To evaluate the number of lines that could be replaced by the library, we investigated nine verification projects. VMM was used for four projects and VMM-like in-house methodology was used for five projects. A total of 489,875 lines of existing SystemVerilog code was investigated. This number of lines included class-based code such as tests, verification components, and verification objects, but excluded module-based code such as design under test (DUT) and top-level test benches. The comment lines and blank lines were also excluded. The preliminary investigation revealed that about 2% of code could be replaced by the library.

One of the main goals of the generic library was to reduce the number of lines by reusing common functions. However, the 2% code reduction fell somewhat short of our expectations. Was this because the library was not diversified enough to cover the wide range of verification functions? Or, was this because the verification work itself intrinsically consisted of a project-specific unique set of codes, and thus no additional common library could be created? To answer these questions, further investigations were made to reveal what the other 98% of code carried out.

A. *Code Breakdown*

We categorized our code into two groups: a fixed-pattern group and a project-specific group. The code categorized into the fixed-pattern group uses fixed style and usually cannot be replaced by the common library. The fixed-pattern group includes compiler directives, interface definitions, type definitions, declarations, constraints, functional coverages, register abstraction layers (RAL), and boilerplate codes. The boilerplate codes are the functions that have to be included to make verification methodology work. The examples include the `copy`, `compare`, `byte_pack`, and `byte_unpack` functions of VMM [15]. The code not classified into the fixed-pattern group is categorized into the project-specific group.

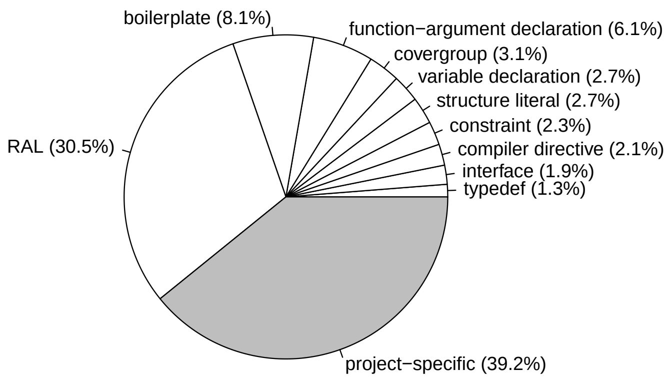

Fig. 4. Breakdown of 489,875 Lines of Verification Code

Interestingly, the investigation results showed that about 60% of the codes were categorized into the fixed-pattern group (Figure 4). If we excluded the fixed-pattern code, the reduction rate rose to about 5%. Considering that most of the library consisted of low-level functions, this number seemed reasonable. Even though the number of fixed-pattern codes took 60% of the entire code, and the library might give little benefit to them, the fixed-pattern codes were relatively easy to develop and could even be generated automatically. As the project-specific codes are the most vital part, this reduction rate seemed satisfactory.

B. *Limitations of the Library*

Although our library can be extended to support more variety of functions, some functions are probably infeasible to develop in SystemVerilog. This is because of the performance and expressiveness of the language compared to other programming languages.[3] Such functions may include processing regular-expressions, image processing, and parsing external format such as JavaScript Object Notation (JSON) [16]. One approach to overcoming this limitation is to use another programming language through DPI. SystemVerilog defines a foreign-language layer only for the C programming language, but with a little effort we would be able to interface the SystemVerilog to C++ as shown in Section III.

[3]The language expressiveness is measured by how simply a concept can be expressed in the language.

The library does not break the language limitations either. For example, the library cannot add a new syntax to the language such as a list comprehension, nor add a new programming paradigm such as a lambda function (Python supports both). One interesting approach to overcome this limitation without modifying the language itself would be to create a library similar to Functional Java [17]. However, we have not yet evaluated the usefulness of such an approach for the design verification field.

C. *Difficulties of Open-Source Projects*

Open sourcing does not magically make a project a success. In his booklet on open-source projects, Fogel [18] mentions that most free software projects fail. This is because they add new sets of complexities, such as deploying a development web site, writing documentation, packaging, and managing contributors who have never met each other. In addition to these common hurdles, there is another restriction specific to verification libraries. Unlike other open-source projects, in which contributors use their own time and software, the verification libraries are usually developed using their employer's resources. Typically, the copyright of such libraries belong to their employer. To avoid potential risks with lawsuits, we have decided not to accept any "pull requests" from individual contributors. This may not be the ideal eco-system, but the users can still make comments on the library and fork the library to extend it by themselves. We hope a social coding site like GitHub would ease these activities. Although we will not accept the source code in itself, we will accept a request for developing a new class or a function. Because the new function might immediately benefit other users, we will try to add it on a regular basis to make the library more useful.

XI. Conclusions

We created a library to handle common verification tasks, and opened it to the verification community. Our library was independent from any verification methodology. One area of future work will be to develop a methodology-dependent library, such as for UVM. Examples include an extension of `uvm_tlm_generic_payload` class, an event waiter that waits for a `uvm_event` with a timeout, and a `report_phase` function that collects the simulation statistics [19].

Python lovers often use the phrase "batteries included" to describe its standard library [20], which covers many utility functions. One reason that coding in Python is fun may be because of this rich set of libraries. We hope that our library may become the starting point for the shared generic library that makes coding fun again in SystemVerilog, too.

XII. Acknowledgment

We gratefully acknowledge Verifore, Inc., who provided the source code of their verification projects for the code analysis, without whose help this work would not have been possible.

References

[1] DVCon 2013 Demographic Totals. Design & Verification Conference & Exhibition. [Online]. Available: http://dvcon.org/sites/stage.dvcon.org/files/files/DVCon2013DemographicTotals.pdf

[2] A. Sarkar, "SystemVerilog FrameWorks^{TM} Scoreboard: An Open Source Implementation Using UVM," in *Design & Verification Conference & Exhibition Proceedings*, 2011.

[3] G. van Rossum, *The Python Library Reference Release 3.3.2*, Python Software Foundation, October 2013.

[4] J. Britt. Index of Files, Classes & Methods in Ruby 2.0.0. Ruby-doc.org. [Online]. Available: http://ruby-doc.org/core-2.0.0/

[5] *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE Std. 1800-2012, February 2013.

[6] GitHub – Build software better, together. GitHub, Inc. [Online]. Available: https://github.com/

[7] *Information Technology – Programming Languages – C++*, American National Standards Institute Std. INCITS/ISO/IEC 14 882-2012, 2012.

[8] Java Platform Standard Edition 7 Documentation. Oracle and/or its affiliates. [Online]. Available: http://docs.oracle.com/javase/7/docs/

[9] Perl Programming Documentation. perldoc.perl.org. [Online]. Available: http://perldoc.perl.org/

[10] *ECMAScript Language Specification*, Ecma International Std. ECMA-262, Rev. 5.1, June 2011.

[11] ClueLib – Sharing Generic Class Libraries in SystemVerilog Makes Coding Fun Again. ClueLogic, LLC. [Online]. Available: https://github.com/cluelogic/cluelib

[12] *Control Functions for Coded Character Sets*, ECMA Std. ECMA-48, Rev. 5, June 1991.

[13] The MIT License (MIT). Open Source Initiative. [Online]. Available: http://opensource.org/licenses/mit-license.php

[14] K. Finley. (2011, June) GitHub Has Surpassed SourceForge and Google Code in Popularity. Say Media Inc. [Online]. Available: http://readwrite.com/2011/06/02/github-has-passed-sourceforge

[15] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog*. Springer, 2005.

[16] Introducing JSON. [Online]. Available: http://www.json.org

[17] Functional Java. [Online]. Available: http://functionaljava.org

[18] K. Fogel, "Producing Open Source Software – How to Run a Successful Free Software Project," 2005.

[19] *Universal Verification Methodology (UVM) 1.1 Class Reference*, Accellera Organization Std., June 2011.

[20] About Python. Python Software Foundation. [Online]. Available: http://www.python.org/about/