Generic Programming in SystemVerilog

Mark Glasser
NVIDIA Corporation
Santa Clara, CA 95050
mglasser@nvidia.com

***Abstract*—Making programs generic, or generic programming, means making programs independent of information about types, sizes, locations, and similar specific information. This requires certain programming styles and language features which avoids coding low-level details. SystemVerilog, with its heritage as a hardware modeling language, has some features for generic programming but is not considered a generic programming language.**

We have developed the SystemVerilog Extension (SVX) library which provides facilities that improve the capability of SystemVerilog to render generic programs. SVX is written entirely in standard SystemVerilog with no reliance on DPI or any other external interface.

Aspects of generic programming are discussed, including removal of assumptions and abstraction.

We give a tour of the library and its facilities and features. Then we discuss concepts of generic programming in terms of SVX. Finally, we compare our work with others who have approached the same topic and describe some future directions for generic programming in SystemVerilog.

***Index Terms*—generic programming, data structures, SystemVerilog, Verilog, verification**

I. Introduction

Anyone who has done any programming at all in any language quickly discovers they are writing programs or parts of programs that are very similar. When one day you need to sort integers and another day you need to sort strings you have the nagging suspicion that there must be a way to write one piece of code that sorts both kinds of things. The algorithm for sorting is the same, whether the type of the thing you are sorting is integer or string. The differences are in type-specific details that are independent of the actual sort algorithm.

Enter generic programming. Generic programming is a style of writing code that enables programs to be used for multiple purposes, such as writing a single program (or function or class) for sorting lists of strings, integers, or imaginary numbers stored as pairs of floating point values. Generic programming involves writing algorithms that are independent of data types, size of objects or structures, memory allocation models, and other low-level details. Generic programs are highly reusable.

To make a program generic you must imbue it with *degrees of freedom*, means of configuring or changing aspects of the program without changing the program itself. The most obvious way is to parameterize things. Changing a parameter can change some aspect of a program without touching the program's code. Another way is to employ abstraction. Implementation details can be hidden behind abstract interfaces. Alternate implementations of the interface can be supplied without disturbing code that uses the interface. The most basic form of generic programming is using variables or named constants instead of literal constants. Another common form of generic programming is using functions as a means of abstraction.

While SystemVerilog supports some means for building generic programs it is not generally thought of as a generic programming language. SystemVerilog provides things like functions and parameterized classes, but lacks other things like parameterized functions, or partial template specialization.

A. Assumptions

All of those low-level details represent *assumptions*. Making a program generic means removing assumptions within the program. An assumption is essentially anything that's hard-coded, usually a value, size, or type. By hardcoding something we are *assuming* that it will never change. An assumption can be quite blatant such as bounds of a for-loop or it can be quite subtle such as the return type of a function. As an example of the latter kind of assumption, consider the following function (in C++):

```
int f(int a, int b) { return a * b; }
```

Function `f()` contains a subtle assumption that the product of `a` and `b` will always fit into an integer without overflowing. Perhaps this is an acceptable assumption for the given application, perhaps not. One way to eliminate the assumption is to have the return type be a `long` instead of just an `int`. Another way is to check number of bits in the arguments and throw an exception if the product would not fit into the return type. Each of these means of eliminating an assumption has consequences in terms of the program structure, and each has a cost. The architectural consequences and cost of eliminating the assumption must be weighed against the risk of leaving it in.

Removing an assumption can be as simple as replacing a literal constant with a named constant or a variable. You need to find a place to initialize the value of the variable and you must take care to ensure that the variable is always used in places where the constant was used before. Often, however, removing an assumption requires more thought about the nature of the assumption and the costs of eliminating it.

It's usually not possible to remove *all* assumptions in a program. However, assumptions that remain should be intended and documented. For example, in building a memory model it is useful to assume bytes are eight bits. Removing that

assumption and allowing bytes to be any number of bits is expensive in terms of code and testing and does not necessarily provide any real value to the user. Perhaps someone may want to create a memory model with 12-bit bytes or 7-bit bytes but those situations are very rare, if they exist at all. So it is reasonable to make the assumption that bytes are always 8 bits. When we document an assumption it becomes a *constraint*. When a user chooses our memory model for an application he can decide whether or not he can live with the constraint that bytes are 8 bits. He can only make that decision of the constraint is documented.

B. Reuse

Reuse is about adapting a piece of code to a new situation. Every degree of freedom represents an assumption that has been replaced with some means of modifying an aspect of the code from outside without modifying the code itself. The degrees of freedom represent ways in which the code can be reused.

As a counter-example of code that is not very reusable, consider a class that implements a stack with a fixed size. For a particular application it's known that the stack will never have more than ten entries. So, a fixed-size stack is acceptable in this case. It meets requirements and the program that uses this stack works correctly in all the tests and in production. In a completely separate application another stack structure is required. This time it's likely that as many as one hundred items will be on the stack at once. Can our stack be reused?

No, not really. The stack class does not meet the requirements of the current application. It will have to be modified to make it meet the new requirements. It does not have a degree of freedom that enables it to be reused in these two applications. Any time you have to modify code you cannot claim you are reusing it. Copy and modify (sometimes referred to informally as cut-and-paste) is not reuse.

The correct thing to do is to build a stack class that has stack size as a degree of freedom. Then copying and modifying would not be necessary. Instead any program that needed a stack could *refer* to an entry in a library.

The more assumptions you can eliminate the more generic is your program. Eliminating assumptions increases the reusability of a piece of code. So, the more generic a program is the more it can be reused.

C. Abstraction

Often, in the world of design and verification we talk about abstraction and abstraction levels. An *abstraction level* is a particular abstraction of data and time used to model systems. A common abstraction level is labeled *RTL*. In RTL data is bit accurate and time is discrete. Terms like RTL and transaction-level have become familiar within the design and verification community as representations of particular abstractions.

When we talk about abstraction in a discussion about generic programming we mean something a bit different. We are talking about hiding implementation details, usually behind and interface. In some cases this will abstract the kind of data that we deal with, but that is not necessarily the intent. The details are still there, they are just not immediately visible.

SVX, which we will discuss in detail coming up, has a vector container. The vector has operations such as read, write, clear, etc. There is nothing magic in the implementation of `vector#(T,P)`, it's based on the SystemVerilog unbound queue. All of the operations available on our vector are also available on the queue built in to the language. So why not just use the built-in queue and not even bother with the vector?

The reason is that the vector container provides a useful abstraction. All of the details of the operations are hidden behind a set of class methods. Details about full or empty containers, first and last objects, copy, clone, compare, etc. are out of view and therefore no longer important. Each operation has a name that reflects what it does. You can think in terms of those operations an not have to think terms of low level constructs. To experienced SystemVerilog programmer it may be obvious that `q[0:$-1];` represents a `pop_back()` operation, but to most it would not. They would have to do some mental gymnastics to realize this little fragment of code represents something more abstract.

Abstraction also enables us to be more consistent. We can provide a consistent interface for vectors, deques, queues, and even trees and graphs. Again, the details are hidden out of view so you only need to deal with the abstraction of the operations and not their implementation.

II. The SVX Library

The SVX Library is a library written in standard SystemVerilog that supports generic programming. It contains two major parts, one for representing data structures, and the other for representing behaviors. The following is a tour of the library. It does not represent complete documentation for the library, only a discussion of the essential features with an emphasis on those facilities that enable generic programming.

III. Structure

SVX provides some data structures in the spirit of C++ STL. These are vector, queue, stack, dequeue, map, and tree. The data structures are containers that hold arbitrary typed objects. Figure 1 shows all of the structural elements in the library[1].

A. Traits Classes

In order to understand SVX containers you first need to understand how traits classes are used in SVX. There are some aspects or *traits* of a data type that the compiler cannot figure out for itself. It may not be able to ascertain the empty object or how to sort (order) objects of that type, for example. We have to explicitly supply this information for the compiler. We do this using a *traits class*.

A traits class, as the name suggests, is a class that supplies information about the characteristics or traits of a type, information that the compiler cannot determine for itself. SVX

[1]Because of the differences between C++ and SystemVerilog there was no attempt to replicate STL or rewrite it in SystemVerilog. It simply was a source of inspiration for a SystemVerilog facility.

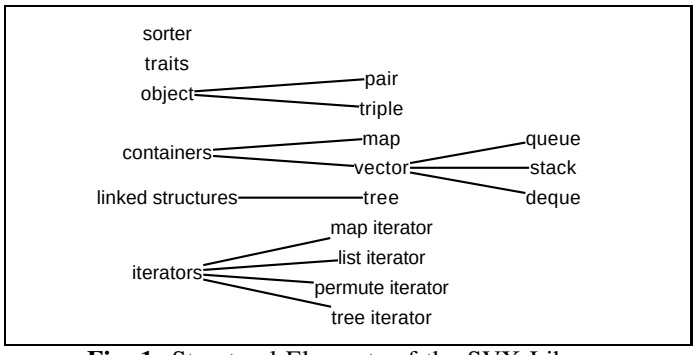

Fig. 1: Structural Elements of the SVX Library

`typedef empty_t`	prop	typename of the empty object
`empty_t`	prop	value of the empty object
`compare`	func	compare two object for less-than, greater-than, or equal
`equal`	func	compare two objects for equality
`sort`	func	sort a collection of objects

TABLE I
TABLE OF TRAITS CLASS MEMBERS AND METHODS

uses traits classes to supply information such as the type of an empty object, the value of an empty object, and compare, equal, and sort functions[2].

SystemVerilog does not support semantics like partial template specialization in C++. So it's not possible to create traits classes that are automatically recognized by the compiler based on their type. Instead, we must explicitly supply a traits class to a data structure.

Since traits classes are type-specific they do not have a base class. Nonetheless traits classes have a specific form which must be adhered to in order to work properly in SVX. The required set of members and methods for a traits class are shown in table I.

Here is a prototypical traits class for some type T. Replace T with the name of a type to create a traits class for a specific type.

```
class T_traits extends void_t;

  typedef T empty_t;
  const static empty_t empty = <empty value>;

  static function bit equal(input T a,
                            input T b);
  endfunction

  static function int compare(input T a,
                              input T b);
    return !equal(a,b);
  endfunction

  static function sort(ref void_t vec[$]);
  endfunction
```

[2]Traits classes in SVX contain behaviors so they technically could be called policy classes. However, they also contain properties so the term traits classes is correct and perhaps equally incorrect. The term traits classes seems most appropriate since the purpose is to describe a type and not specifically for providing interchangeable algorithms.

```
endclass
```

Implementations for `equal()` and `compare()` are required. An implementation for `sort()` is optional. If you do not have a need to sort objects of the type associated with the traits class then an implementation of `sort()` is not necessary. `Sort()` can be implemented in one of several ways. For scalar types the built-in sort command can be used. Or you can use the built-in generic sorting class. The built-in sorting class is generic because it does not make any assumptions about the type of object it is sorting. It is parameterized with the type of object to be sorted and it relies on the compare() function in the traits class to determine relative ordering of objects.

We'll look at how traits classes are used in the next section, III-B, on containers.

B. Containers

The organization of the container classes are shown in 2. The primary base class is `container`. Most of the operations on containers require some knowledge of the type of the object in the container so there are not a lot of virtual functions in the base class. In fact, there are only two — `size()` and `clear()`. `Size()` returns the number of items in the container, and `clear()` empties the container — that is, it removes all of the objects in the container.

The class `typed_container` is a base class that knows about the type of the object held in the container. It takes two paramaters, `T`, which is the object type, and `P`, which is the traits class for type `T`. It contains no additional methods and only one property, `m_empty` which holds the empty object for type `T`.

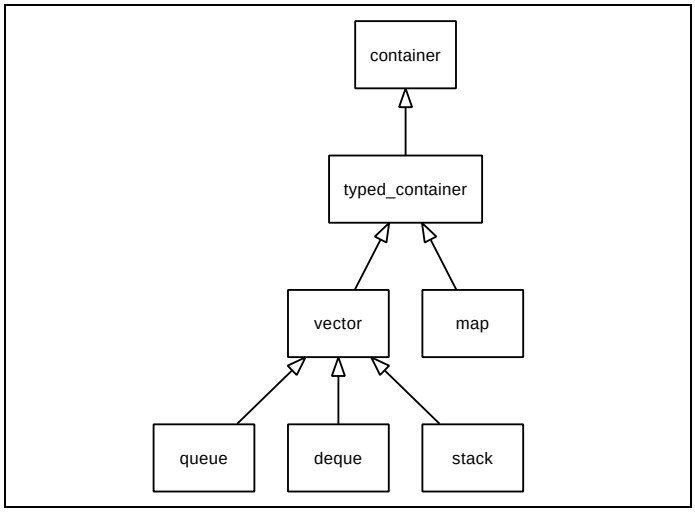

Fig. 2: Relationships of Container Classes

`Vector` is an unsized list of objects of type `T`. The objects can be referenced by index. Because the underlying structure is a SystemVerilog queue, the head and tail (front and back) of the vector can be accessed, though the textttvector API does not provide this capability. Derived objects, `deque`, `stack`,

and queue do provide access to the head and tail based on the semantics of the particular data structure.

The map container maps a key object to a data object. Keys must be unique. Objects can be inserted, deleted, and looked up by key.

Containers are parameterized by the type of thing they contain and the relevant traits class. The following declaration of a vector takes two parameters.

```
vector#(string, string_traits) v;
```

The first parameter specifies that the vector is a vector of strings. The second parameter is a traits class for strings. Providing the traits class as a parameter gives the vector class all the information it needs to perform all of the interface operations in a type-safe and type-correct manner.

One of the operations that is available for vectors is read(). Read() takes an index as an argument. If the index is out of bounds of the vector then read() is expected to return an empty element. For a string the empty element is an empty string, which syntactically is a pair of double quotes(""). The traits class tells the vector class for a string the empty object is "". For a vector of integers, for example, we would use an int_traits class as the second parameter. When we call read() with an out-of-bounds index then a zero is returned. For integers, the empty object is 0.

Traits classes make the container classes truly generic. Using traits classes we can configure containers to operate on any type without restriction. Any type-specific could that would need to be in the container classes can be instead in the traits classes.

C. Iterators

Iterators are class objects that attach to containers and provide a means to easily traverse the containers and access objects sequentially. Iterators gives us a way to access containers in an abstract and generic way.

SVX provides several kinds of iterators:

- List iterator — iterate over vectors, queues, stacks, and dequeues.
- Map iterator — iterate over maps.
- Permute iterator — iterate over permutations of a list.
- Tree iterator — iterate over items in a tree.

Each kind of iterator has four types, each of which implements a different interface — forward, backward, bidirectional, and random — for a total of sixteen different iterators. The bidirectional iterators implement both the forward and backward interfaces.

The forward interface provides a means to sequentially traverse objects from first to last. The particular object that is first and the order of the objects is defined by the container.

```
interface class fwd_iterator
  extends iterator_base;
  pure virtual function bit first();
  pure virtual function bit next();
  pure virtual function bit is_last();
  pure virtual function bit at_end();
endclass
```

The backward interface is similar to the forward iterface except that it provides a means for traversing objects sequentially from last to first.

```
interface class bkwd_iterator
  extends iterator_base;
  pure virtual function bit last();
  pure virtual function bit prev();
  pure virtual function bit is_first();
  pure virtual function bit at_beginning();
endclass
```

The random interface provides a means to randomly select an object within the container. It also has functions for managing the random number generator seed in order to maintain random stability in applications where that is required.

```
interface class random_iterator
  extends iterator_base;
  pure virtual function bit random();
  pure virtual function void set_seed(int seed);
  pure virtual function void set_default_seed();
endclass
```

The iterator_base base class provides a skip() function for all interfaces. Whereas the next() and prev() functions move the state of the iterator forward or backward one item, skip() will move the iterator state forward or backward one or more items. A positive argument to skip() will move the state forward, a negative argument will move the state backward. The implementation of skip() is dependent on the nature of the interface. For example, for iterators that implement fwd_iterator, skip() will only allow the state to be moved forward and not backward; similarly, iterators that implement bkwd_iterator the state can only be moved backward.

```
interface class iterator_base;
  pure virtual function
    bit skip(signed_index_t distance);
endclass
```

In no case, no matter which iterator function is used, can the state be moved beyond the beginning or end of the container.

Iterators, like their counterpart containers, are parameterized using traits classes.

IV. Behavior

SystemVerilog provides a syntactic way to define concurrent behaviors via the fork/join construct (and its cousins fork/join_none and fork/join_any). This makes it easy to identify concurrent behaviors in code, but difficult to manage them. Using only the SystemVerilog constructs concurrent processes are static, they can only be executed when the locus of control passes through the process definition(s).

SVX has loosened the bonds of processes making them objects that can be stored, passed through function arguments, and started and stopped at any time.

The primary elements of a behavioral part of the SVX library are behaviors and processes. A behavior is an abstraction of a task and function. Behaviors can be executed in a blocking or nonblocking fashion. In order to execute a behavior as nonblocking it must be attached to a process. A process facilitates the concurrent execution of a behavior. Processes are mainly used for executing task behaviors, since

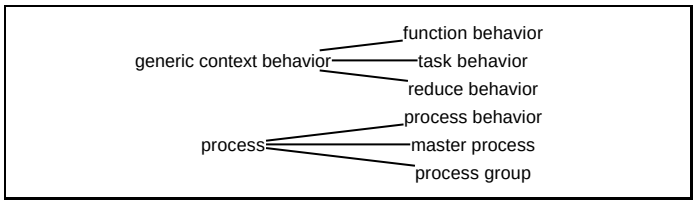

Fig. 3: Behavioral Elements of the SVX Library

task behaviors may consume time. However processes do not distinguish between task and function behaviors and can be used to execute either. The SVX classes for behaviors and processes and their relationships are shown in figure 4.

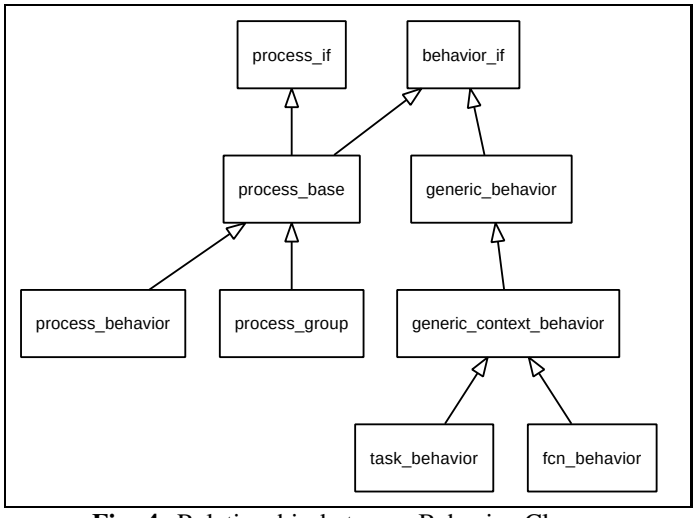

Fig. 4: Relationship between Behavior Classes

A. Functions and Tasks

`Process_if` is an interface class with virtual functions for executing a behavior. `Nb_exec` executes a behavior in a nonblocking manner, that is without consuming a delta cycle; `exec` executes behavior in a blocking manner. A blocking behavior may or may not consume time.

```
interface class behavior_if;
  pure virtual task exec();
  pure virtual function void nb_exec();
endclass
```

The class `generic_context_behavior` introduces the concept of a *context*. A context is a data object that supplies data to a behavior. Or, stated another way, a behavior operates upon a context. A behavior may alter the state of a context object. The parameter to `generic_context_behavior` defines the type of the context object for a family of behaviors which are derived from `generic_context_behavior`. This class provides some methods for executing behaviors with a context and managing a context object.

```
virtual class generic_context_behavior#(type T=int)
  extends generic_behavior;

  protected T c;

  virtual task exec();
  virtual function void nb_exec();
  virtual function void bind_context(T cntxt);
  virtual task apply(T cntxt);
  virtual function void nb_apply(T cntxt);
  virtual function T get_context();
endclass
```

`Exec` and `nb_exec` are implementations of the virtual functions defined in the parent class `behavior_if`. `Bind_context()` stores a context object so it can be accessed be `exec()` and `nb_exec`. `Apply()` is equivalent to `bind()` followed by `exec()` — it first binds the context to the behavior and then executes the behavior. `Nb_apply()` works like `apply()` except that it executes the behavior in a nonblocking manner.

The `task_behavior` and `fcn_behavior` classes contain the behavior to be executed. `Task_behavior` has a virtual method `tsk()` and `fcn_behavior` has a virtual method `fcn` which must be implemented with a task or function behavior, respectively. We'll call these the behavior methods. To create a behavior derive a class from the appropriate base class for a task or function behavior and implement the behavior method. For example:

```
class my_task_behavior
  extends task_behavior#(context_type);

  task tsk()
    // task behavior goes here
  endtask
endclass
```

`Generic_context_behavior` contains a protected variable `c` which is the context. To access the context you can simply refer to `c`. Putting all the parts together, we can demonstrate a (trivial) behavior that simply executes a delay and the delay comes from the context.

```
class delay extends task_behavior#(int);
  task tsk();
    #c
  endtask
endclass
```

To execute the behavior we must instantiate it and execute it via one of the execution methods. In the following example we will use `apply()` to bind the context and execute the task.

```
delay d = new();
d.apply(5);
d.apply(20);
```

The two `apply()` calls in our example execute in blocking mode. Time will advance 5 time units after the first call and 20 more after the second.

We can alter our previous example so that the tasks execute in nonblocking mode.

```
delay d = new();
d.nb_apply(5);
d.nb_apply(20);
```

In nonblocking mode time does not advance after each call. Instead the tasks are launched independently and will finish independently. This is equivalent to executing processes using `fork/join_none`.

B. Processes

The process classes provide fine-grained process control for blocking behaviors similar to SystemVerilog.

```
interface class process_if;

  pure virtual function void suspend();
  pure virtual function void resume();
  pure virtual function void kill();
  pure virtual function bit is_done();
  pure virtual task await();

endclass
```

Processes are bound to behaviors and provide the fine-grained control over the execution of the bound behavior.

We will look at some applications for behaviors in section VI-D.

V. Other Utilities

The permutation iterator and the lexer are two utilities in the SVX library that aid in writing generic programs.

A. Permutations

The permute iterators operate on permutations of a vector. They provide a means to walk through some or all of the permutations of a list and randomly select permutations. Permutations have interesting applications in verification. Consider a device that has some set of operations which must be verified. To thoroughly verify the operations you must show not only that each one works correctly, but also that each works correctly no matter which operation precedes or follows it. The best way to do this is to run the operations in some order and all of the permutations of the order. If the device has three operations, A, B, and C then we need to run all the orderings of three operations.

```
ABC, ACB, BAC, BCA, CAB, CBA
```

The number of permutations of a set is $n!$ (n-factorial) which is defined as $n \times (n-1) \times (n-2) \times ... \times 1$. A list of three items has $n! = 6$ permutations.

For a large list of operations it may not be feasible to run all of the permutations. A list of 10 items, for example, has $10! = 3,628,800$ permutations. Instead of running all of permutations you could randomly select some percentage of the total permutations.

The permutation iterator works much like the other iterators. In fact, they implement the same iterator interface classes for forward, backward, bidirectional, and random iterators.

The base class, `permute_iterator_base` provides some additional functionality for dealing with permutations. Like the other iterators the permutation iterators also bind to a vector. Behind the sceneds they create a `permutation vector` which contains references to items in the original vector however, in different orders. Because references are used objects are not copied.

```
class permute_iterator_base#(type T=int,
                             type P=void_traits)
  extends typed_iterator#(T,P);

  virtual function void bind_vector(vec_t v);
  virtual function T get_nth(index_t n);
  virtual function longint get_permutation_index();
  virtual function vector#(int, int_traits)
                      get_permutation_vector();
  virtual function bit skip(signed_index_t distance);
endclass
```

`Bind_vector()` binds a vector to a permutation iterator. It also initializes the internal permutation vector and other state information. The initial ordering of the bound vector is the 0^{th} permutation. The bound vector is never modified, only the internal permutation vector is modified to reflect a new permutation. If a vector is already bound you can use `first()` to re-initialize the internal state to the 0^{th} permutation. `Next()` will advance the internal state to the next permutation. `Is_last()` and `at_end()` can be used to determine if the iterator state has reached the end of the permutations.

The base class functions `set()` and `get()` have no meaning for the permute iterators. Instead, `get_permutation_index()` returns the index of the current permutation, and `get_nth()` returns the n^{th} item of the current permutation.

B. Lexer

Surprisingly, SystemVerilog coding often involves a fair amount of text processing. Unfortunately, the SystemVerilog language does not provide much in the way of text processing facilities. In their respective works, [1] and [2] have each addressed text processing. We did not try to replicate their work. Instead, we focused on an aspect of text processing that is often overlooked, namely parsing. SVX provides a lexical analyzer that can be used to extract tokens from a stream of characters. The interface for the lexer is shown here:

```
class lexer_core;
  function void start(string _s);
  function string get_lexeme();
  function token_t get_token();
  function token_descriptor get_token_descriptor();
endclass
```

`Start()` binds the lexer to a string of characters of arbitrary length. To parse an input stream, repeatedly call `get_token()` until an `EOL` (end-of-line) token is reached. `EOL` does not necessary mean the end of a line per se, it means the end of the input string. The input string may contain whitespace characters such as tab and newline. `get_lexeme()` returns the string associated with a token. This is useful to get the name of an identifier, for example. `get_token_descriptor()` can be called after retrieving a numeric token. It returns additional information about the value and radix of a number.

A typical parsing loop looks like this:

```
lexer_core lex = new();
lex.start(s);

do begin
 token = lex.get_token();
end while(token != TOKEN_EOL);
```

Of course, after the call to `get_token()` code can be inserted to process the tokens. Table II has a list of tokens recognized by the lexical analyzer.

The lexical analyzer is useful for parsing command line options or input files. It also aids the cause of generic programming by providing a means where information of various sorts can be recognized programmatically (i.e. parsed) instead of hardcoded.

TOKEN_AMPERSAND	TOKEN_GREATER_EQUAL	TOKEN_PLUS
TOKEN_AT	TOKEN_GREATER_THAN	TOKEN_POUND
TOKEN_BACKTICK	TOKEN_ID	TOKEN_QUESTION
TOKEN_BANG	TOKEN_INT	TOKEN_RIGHT_PAREN
TOKEN_CARAT	TOKEN_LEFT_PAREN	TOKEN_SEMI
TOKEN_COLON	TOKEN_LESS_EQUAL	TOKEN_SLASH
TOKEN_DOLLAR	TOKEN_LESS_THAN	TOKEN_STAR
TOKEN_DOT	TOKEN_LOGIC	TOKEN_STRING
TOKEN_EOL	TOKEN_MINUS	TOKEN_TILDE
TOKEN_EQUAL	TOKEN_OFF	TOKEN_TIME
TOKEN_ERROR	TOKEN_ON	
TOKEN_FLOAT	TOKEN_PERCENT	

TABLE II
TABLE OF TOKENS

VI. USING SVX

The purpose of writing generic code is to streamline code and make the underlying logic readily accessible and not hidden amidst a jungle of low-level details. Here are some examples that rely on the facilities available in SVX.

A. Traversal

Let's first start with an idiom for traversing a vector:

```
vector#(int, int_traits) v;
int t;
list_fwd_iterator#(int, int_traits) iter = new(v);

iter.first();
while(!iter.at_end()) begin
  t = iter.get();
  // do something with the element
  iter.next();
end
```

One thing to notice is that there are no constants in the code! The logic is easily visible without relying on numbers of any sort. The begin and end points of the vector are abstracted behind `first()` and `at_end()`. The iterator has the same parameters as the vector it will traverse. Supplying the vector, `v`, as an argument to the constructor binds the vector to the iterator.

Now let's look at code for traversing all of the elements in a map. Our example map maps strings to objects.

```
map#(string, object, object_traits) m;
object t;
map_fwd_iterator#(string, object,
                  object_traits) iter = new(m);
iter.first();
while(!iter.at_end()) begin
  t = iter.get();
  // do something with the element
  iter.next();
end
```

The code for traversing a map is the same as for traversing a vector. We can make the code look identical because we have abstracted out the details. The thing that remains is the traversal logic. Of course, the implementations of `map_fwd_iterator#(K,T,P)::first()` and `list_fwd_iterator#(T,P)::first()` are quite different. However, at the user level, the level you are concerned with as a user of the SVX library, the `first()` functions do essentially the same thing. Their semantics are similar — to reset the iterator to the first element.

I'm sure by now the reader can guess what the code will look like to traverse a tree. Yes, you are right! It looks the same as the code for traversing a list and a map.

```
tree m;
tree t;
tree_fwd_iterator iter = new();
iter.bind(m, PREORDER);

iter.first();
while(!iter.at_end()) begin
  t = iter.get();
  // do something with the element
  iter.next();
end
```

The main difference is that when you bind the tree to the iterator you must specify the order in which the tree will be traversed.

B. Memory Management

Language defined structures such queues and associative arrays in SystemVerilog have scopes that are syntactically defined. This means that the scope of a queue or an associative array is only within the scope where it is declared. Further, these structures are allocated by the compiler as the flow of control enters the appropriate scope and not via the new() method.

By contrast, the structures in SVX, vector, queue, stack, etc. are class objects and thus come into existence when `new()` is called. These objects can be passed as class handles through argument lists.

A queue defined using SystemVerilog syntax looks like this:

```
int q[$];
```

Queues declared in this manner can be assigned to each other. However doing so entails copying the entire queue. This is true for arrays and associative arrays as well. SVX queues (of the same type) can also be assigned to each other. Since they are class objects only the handles are assigned and the entire queue is not copied. The map and vector structures support `copy()` and `clone()` operations for those occasions where the entire structure must be replicated. SVX structures can efficiently be stored or passed through argument lists without unnecessary copying.

C. Trees

Trees in SVX are not containers in the same sense as vectors, queues, stacks, deques, and maps. Instead `tree` is a base class from which you can create your own hierarchical object. Creating a hierarchy of any sort is as simple as extending the `tree` class.

We'll use a hierarchical memory map as a way to demonstrate how to use trees. To make a hierarchical memory map you could define an object that represents a portion of an address space. You can provide tree semantics simply by extending your address space object.

```
class addr_map extends tree;
  addr_t base_addr;
```

```
  addr_t limit_addr;
endclass
```

Each tree node has a name, a parent and zero or more children. You can create your hierarchy by creating a root node and adding children as desired.

```
addr_map root, slave, mem, sys;
addr_map = root;
root = new("root", NULL);
sys = new("sys_bus", root);
root.insert(sys);
mem = new("mem_bus", root);
root.insert(mem);
slave = new("first_slave", sys);
sys.insert(slave);
slave = new("second_slave", sys);
sys.insert(slave);
// etc...
```

Nodes in a tree can be located by name. The `find()` function takes apart the dot-separated name using the lexical analyzer. A recursive algorithm locates each child and sub-child until either a complete match is found or a child is not located.

You can use the tree traversal functions to create your own search.

```
tree m;
tree t;
tree_fwd_iterator iter = new();
iter.bind(m, INORDER);

iter.first();
while(!iter.at_end()) begin
  t = iter.get();
  if(match_tree_node(t)
    break;
  iter.next();
end
```

`Match_tree_node()` is a user-written function that matches a tree node against user-defined criteria. Instead of a `break` upon locating a matching node, you could instead store the node in a list and create a set of all matching nodes.

D. Using Behaviors

Now that we have behaviors that are contained in objects, we can store them, start and stop them at any time, apply them to many contexts, and apply many behaviors to a single context.

For example we can apply a single behavior to multiple contexts stored in a list:

```
some_task_behavior b = new();
iter.first();
do begin
  t = iter.get();
  b.apply(t);
end while(!iter.at_end());
```

In a verification context this could perhaps be used to randomize a set of objects in a list. Or it could be used in a checker to validate correctness of a set of objects.

Or, we can store behaviors in a list and apply each one to a single context.

```
iter.first()
do begin
  b = iter.get();
  b.apply(t);
end while(!iter.at_end());
```

This is similar to a callback model where behaviors are ”called back” thus avoiding coding them inline. This provides a degree of freedom around the specific behavior that is executed at a point in the program.

A clock-generator for multiple clocks is an application for concurrent behaviors. Consider a set of clock-generator behaviors stored in a list. Each one is connected to a virtual interface which in turn is connected to an RTL clock pin. A clock descriptor defines the frequency and other aspects of each clock’s behavior.

```
class clk_behavior#(int unsigned N=1)
  extends task_behavior#(clk_descriptor#(N));
  task tsk();
    wait(c.start_event.triggered);
    forever begin
      c.ckif.clk[c.clk_index] <= 0;
      #c.time_lo;
      c.ckif.clk[c.clk_index] <= 1;
      #c.time_hi;
    end
  endtask
endclass
```

Executing the clocks involves creating an instance of the clock behavior and initiating a process for each clock description.

```
virtual task exec();

 process_behavior#(clk_descriptor#(N)) proc;
 iter_t iter = new(clk_vector);

 clk_procs = new();

 iter.first();
 while(!iter.at_end()) begin
   beh = new();
   proc = new(beh);
   proc.bind_context(iter.get());
   clk_procs.add_process(proc);
   iter.next();
 end

 clk_procs.exec();

endtask
```

Now, we can use the fine-grained process control to start and stop the entire group. To stop all of the clocks we call `suspend()`; to start them running again we call `resume()`. `Kill()` stops all the clock processes. Individual clock processes can similarly be suspended and resumed. This gives our testbench a high degree of control over the device clocks.

E. Structure Composition

The availability of predefined structures makes it easy to combine them in arbitrarily complex ways. You can create lists of trees or trees of lists or maps of lists of trees, the possibilities are endless. Let’s look at some examples.

```
vector#(tree, tree_traits) v_of_t;
map#(string, queue#(object, object_traits),
     class_traits) m_of_q;
```

The first item above, `v_of_t`, is a vector of trees. Each entry in the vector is a root node to a tree of arbitrary size. We just have to provide a `tree_traits` traits class so that the tree objects can be managed properly within the vector.

The second object, m_of_q is a map of queues. It maps strings to queues of objects. To make this work we need two traits classes, one for objects and one for the queue of objects. For the latter we used class_traits which comes in the SVX library. This traits class is used as a general purpose traits class for any kind of class type. It assumes we don't need to sort these classes, and comparison is only for equality of handles. Of course, if you require more sophisticated behavior you can replace class_traits with your own traits class.

To make a tree of vectors, for example, you would have to create a node type extended from tree that contains a vector.

```
class vector_node extends tree
  vector#(string, string_traits) v;
endclass
```

You can use all of the tree traversal and search functions to access modes in the tree.

VII. Comparison with Other Work

Other authors have addressed the issue of generic programming in SystemVerilog and building extension libraries. We'll compare SVX with two other SystemVerilog libraries, *Svlib* [1], written by Jonathan Bromley and André Winkelman, and *Cluelib* [2] written by Keisuke Shimizu.

A. Svlib

Svlib is a utility library as opposed to a library for generic programming. The authors did not specifically address the issue of generic programming. Svlib contains facilities for dealing with files, regular expressions, and string manipulations. It also has a DOM (document object model), a data structure that can be populated from an external YAML file. The DOM is intended as a way to cleanly represent configuration information to be used throughout the SystemVerilog program.

The focus of svlib is on smoothing over platform and language issues for a variety of utilities. Some of things in svlib address the issue of abstraction. For example, the macro `foreach_enum abstracts away the details of writing a loop over the members of an enumerated type. The Simulator provides a means to programmatically access information about the underlying simulator environment by abstracting away the details of the specific simulator.

B. Cluelib

Cluelib takes a different approach to generic programming than SVX. Instead of using traits classes the library requires users to pass information about data types through function arguments.

Users of Cluelib are forced to make a number of assumptions, something that is antithetical to the concepts of generic programming. The library offers both a packed and unpacked array object, requiring the user to choose one or the other. Also, the packed array, unpacked array, and queue classes each require either a SIZE or WIDTH parameter which is used to bound the structure. In the case of the queue the SIZE parameter is optional, used only when converting to a packed or unpacked array. In practice, if you want to take advantage of the conversions to packed or unpacked arrays you will have to supply a fixed SIZE.

In SystemVerilog multiple instances of parameterized objects with different values for parameters are not assignment compatible. Consider the following two queue declarations using the Cluelib library where C is a class type:

```
queue#(C) a;
queue#(C, 10) b;
```

Even though both queue contain objects of type C they are different types and cannot be used interchangeably through assignment or passing as function arguments.

The deque class offers a bit more generic behavior. Not requiring a size parameter, it's not bounded. Since all deques take only one parameters, the type of the object contained in the deque, all deques that contain the same type are assignment compatible. The deque does not offer random access like the packed and unpacked arrays.

Cluelib takes a different approach to dealing with empty objects. Let's compare deque#(T)::get_first() in the Cluelib library and vector#(T,P)::read() in SVX. First the Cluelib function:

```
virtual function bit get_first( ref T e );
  if ( is_empty() ) begin
     return 0;
  end else begin
     e = q.pop_front();
     return 1;
  end
endfunction: get_first
```

Note that the value we are extracting from the data structure is passed as a ref argument. The return value of the function is used to report success or failure of the operation. In the case where the data structure is empty e will have whatever value it had when the function was called. You have to check the return value of the function to see if you retrieved something interesting. The call must always be followed by a check of the return value. For example:

```
deque#(string) q;
string t;
if(!q.get_first(t))
  t = "";
```

Now the SVX function:

```
function T read(index_t idx);
  if(idx >= size())
    return m_empty;
  return m_vector[idx];
endfunction
```

Read() does not return a success or fail flag. Instead it returns m_empty if there is nothing at the index passed in as an argument. m_empty is obtained from the traits class supplied for vector#(T,P)[3]. Depending on the application it *may* be necessary to test the return value for empty value. For some applications an empty value may be just fine and this check is not required.

```
vector#(string, string_traits) v;
string t;
t = v.read(3);
```

[3] M_empty should be P::empty. However the compiler that the author was using to build and test the code balked at this construct. The author found it necessary to assign P::empty to a local class variable

```
if(t == "")
  ...
```

Much of Cluelib is devoted to string processing — accessing and modifying strings or parts of strings. This is something that is sorely needed for SystemVerilog programmers and is not addressed in SVX except by the lexical analyzer.

VIII. Future Work

The next planned addition to SVX is a structure for representing directed acyclic graphs (DAGs). This is a natural extension to the data structures currently available in SVX. The combination of behaviors and graphs can be used in verification applications for driving complex stimulus and tracking state transitions in a DUT. Petri net semantics can be applied to graphs. A shared arbiter object would provide some more control over concurrency, providing a way to create very complex scenarios that are self-managed via the arbiter.

Having an extension library available for SystemVerilog would greatly improve the productivity of programmers and the overall quality of programs. The author would like to see an extension library for SystemVerilog become integral to SystemVerilog tools in the same manner as STL has for C++.

SVX and the other two libraries mentioned in this paper, [1] and [2], have some overlap and also each has some unique features. It would be worthwhile to consider combining all three, taking the best of each, to create a powerful, general purpose extension library for SystemVerilog that provides important structural and behavioral abstractions.

IX. Conclusion

Generic programming is a powerful set of techniques for building transparent, robust, and reusable code. While SystemVerilog has traditionally *not* been thought of as a language for generic programming, generic programs can be written in SystemVerilog – albeit with some help.

The SVX library, a library coded entirely in standard SystemVerilog, provides many facilities for writing generic programs. It introduces the notion of traits classes in SystemVerilog and provides a comprehensive set of data structures and behaviors that can be used to build high quality generic programs. In addition to simply providing some nice utilities, the library enables users to write more abstracted, and thus more reusable, more transparent, and more robust programs.

Acknowledgment

The author would like to thank his employer, NVIDIA, and Anshu Nadkarni for their support of the development of SVX and the production of this paper.

References

[1] J. Bromley and A. Winkelmann, “Systemverilog, batteries included: A programmers utiloity library for systemverilog,” in *DVCon 2014*. Accellera, 2014.

[2] K. Shimizu, “Sharing generic class libraries in systemverilog makes coding fun again,” in *DVCon 2014*. Accellera, 2014.

[3] *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*, IEEE Standards Association, 2012.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.

[5] D. R. Musser, G. J. Derge, and A. Saini, *STL Tutorial and Reference Guide*, 2nd ed. Addison-Wesley, 2001.

[6] R. Sedgewick, *Algorithms in C*. Addison-Wesley, 1990.