SystemVerilog Extension Library (SVX)
Version 1.0

Mark Glasser

March 12, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The SystemVerilog eXtension Library (SVX) provides a collection of parameterized general purpose utilities that aid in the construction of SystemVerilog programs. These utilities extend SystemVerilog in that they provide some convenience features and additional capabilities that are not readily available in the language.

The SVX library has several sections. These include:

Containers. Basic data structures for holding objects of various kinds.

Iterators. Means for traversing containers.

Linked Structures. Data structures constructed from linked nodes, such as trees and graphs.

Behaviors. An abstraction of the SystemVerilog process model.

The library is written entirely in SystemVerilog, relying only on native language features. The library uses the version of SystemVerilog described in the standard IEEE-1800-2012.

Chapter 2

Containers

The SVX library provides a collection of general purpose parameterized containers. A container is a data structure that holds elements of some type, defined by a class parameter.

2.1 Policies

A weakness of SystemVerilog is its inability to allow scalars and aggregate objects to be easily co-mingled in a generic way. You must know whether the object you are dealing with is a scalar or not. Different operators apply. for example, the simple boolean expression

```
(A == B)
```

will easily tell you whether or not the two scalars A and B have the same value. However if A and B are class objects then the result of `A == B` only tells you if A and B point to the same object. If A and B point to different objects the `==` operator will not tell you if the contents of those objects are equivalent.

This presents a problem when building a library of generic classes that can be used truly with any type. To illustrate the problem consider the case of the empty object. The `pop()` method of the stack class must return an empty object if the stack is empty. What should the empty object be? If the stack is defined as `stack#(int)` then a zero is a good choice for for an empty object. If the stack is defined as `stack#(some_class)`, where `some_class` is a class type, then the empty object should be `null`. A zero make no sense in this case.

Our solution is to use policy classes to define characteristics and behaviors of a type that are needed by the library. The policy class defines what the empty object is for a type, whether or not objects of that type can be compared for equality, whether or not objects of that type can be compared for relative magnitude, and so forth.

Schematically, a policy class looks like this:

SystemVerilog Type	Policy Class
int	`int_policy`
int unsigned	`int_unsigned_policy`
longint	`long_int_policy`
longint unsigned	`longint_unsigned_policy`
long long int	`long_long_int_policy`
long long int unsigned	`long_long_int_unsigned_policy`
real	`real_policy`
string	`string_policy`
object	`object_policy`

Table 2.1: Table of built-in type policies

```
class <type>_policy extends void_t;

  typedef <type> empty_t;
  const static empty_t empty = <empty_value>

  static function bit equal(type a, type b);
  endfunction

  static function int compare(<type> a, <type> b);
  endfunction

endclass
```

Where `<type>` is the name of some type. The type can be a scalar type such as `int` or `float`, or it can be a user-defined class type.

The `empty_t` type defines the type of the empty object and the `empty` member contains the empty value. The function `equal()` asks whether or not two object os this type are equivalent. Finally, the `compare()` function returns 0 if the two arguments are equal (as defined by the `equal()` function). It return a value less than zero if $a < b$ and a value greater than zero if $a > b$.

A policy must supply an empty type and value. Implementations of `equal()` and `compare()` functions must be present. However, they may be empty. The return value for empty implementations should be zero. Table 2.1 is a list of all the built-in policy classes.

The *object* policy is a general purpose policy for class types. To create a policy for a user-defined type you can use `object_policy` as a template to build a policy class.

2.2 Containers

The SVX library contains the following set of containers:

vector. An extensible vector object which can be accessed by integer index.

queue. A first-in, first out structure

stack. A last-in, first out structure

deque. A queue that can be pushed and popped from the front and the back

map. Maps keys to objects.

singleton map. A map that can only be instantiated once.

The inheritance hierarcy of the containers is shown in figure 2.1. The class `container` is the base for all containers. Its interface is very simple, providing only a way to find the size of the container (i.e. the number of elements it contains) and the ability to clear the container. The `typed_container` class provides the element and policy types.

The two primary containers are the map and the vector. The vector is a linear list that can be accesed by a non-negative integer index. The map maps keys to elements and elements can be accessed by key. The queue, stack, and deque are derived from vector. Each of these provides all of the semantics of the vector plus additional structure-specific semantics.

2.3 Vector

The vector is a linear list of elements indexed by a non-negative integer. Like all containers the `vector#()` class has two parameters, the type of the elements held in the vector container and a policy class that matches the element type.

```
class vector #(type T=int, type P=object_policy)
  extends typed_container #(T,P);
```

Parameter T represents the type of the vector element and parameter P is the policy class.

The underlying structure of the `vector#()` class is the SystemVerilog queue:

```
typedef T vector_type[$];
protected vector_type m_vector
```

The advantage of using `vector#()` over plain SystemVerilog queues is that you can dynamically create it using `new()`. The vector serves as a base class for other linear containers and thus forms a consistent API for accessing all linear containers.

Data Structure Interface

The data structure interface is for putting items into the vector and retrieving them. A group of append fuctions lets you add a single item or another vector to the end of the current one.

`function void extend(size_t sz)`

Allocate a number of elements in the vector. The elements are added sequentially to the end of the vector.

`function void write(index_t idx, T t)`

Change the value of an element in the vector. The index (idx) must be a non-negative integer. If the index is out of bounds of the array additional empty elements will be added to accomodate the new element.

`function T read(index_t idx)`

Retrieve an element from the array. If the index (idx) is out of bounds of the array the empty element will be returned.

`function void appendv(vector_type v)`

Append a constant vector to the end of this one.

`function void appendc(T t)`

Apapend a single item to the end of the vector. The size of the vector increases by one.

`function void append(this_t v)`

Append another vector to the end of this one.

`function size_t size()`

Return the number of elements currently in the array, including empty elements.

Utility Interface

`function void clear()`

Remove all of the items in the vector. The size becomes zero.

`function void copy(this_t vec)`

Copy the contents of another vector into this one. The original contents of the vector are cleared before the copy starts. When the copy completes this vector is a true copy of the vector passed in as an argument.

`function this_t clone()`

Create a clone of the vector. Allocate a new vector of the same type as this one and copy the contents into the newly allocated vector. The copy is a shallow copy.

`function int compare(index_t idx, T val)`

Compare a single item in the vector with an item passed in as an argument. This is a convenience feature that is useful for searches.

`function bit equal(this_t v)`

Determines if this vector is equivalent to another one of the same type passed in as an argument. The equal function in the policy class is called repeatedly to compare pairs of items with the same index in each vector. If the vectors are not the same size they are consiered to be not equal.

`function void sort()`

Uses the built-in vector sort function to sort the vector.

2.4 Map

Data Structure Interface

`virtual function T get(KEY key)`

Return the element mapped to the argument `key`. If there is no element mapped to `key` then return empty value.

`virtual function bit insert (KEY key, T item)`

Add a new element the structure and map it to `key`. If there already exists an item mapped to `key` it will be overwritten. In that case function will return one, otherwise it will return zero.

`virtual function bit delete (KEY key)`

Remove the element from the structure mapped to `key`.

`virtual function size_t size()`

Return the number of elements currently in the map.

`virtual function bit exists (KEY key)`

Utility Interface

`virtual function bit exists (KEY key)`

Returns one if an element is present that is mapped to `key`.

`virtual function void clear()`

Remove all the elements from the map. The size of the map becomes zero.

`function void copy(this_t rhs)`

Copy all of the elements of the map passed in through the argument `rhs` into this map. The elements in this map are cleared before the copy begins. When the copy completes the two maps have the same elements. The copy is shallow.

`function this_t clone()`

Allocate a new map whose type is the same as this one and copy all of the elements into the new map. The two maps will be identical in structure and contents.

`virtual function int compare(KEY key, T val)`

Compare a single item in the map with one passed in as an argument. The element mapped to `key` is located and compared with the argument element. The comparison is done using the `compare()` function in the policy_class.

`virtual function bit equal(this_t m)`

Compare this map with another one of the same type to determine whether or not they are equivalent. Two maps of the same type are equivalent if they have the same number of elements, and the elements in each map with the same KEY value are equal. The equivalence of two elements is determined by the equal() function in the policy class.

2.5 Singleton Map

`protected function new()`

For a singleton, the standard constructor is not used. It is protected so that it is not visible to users. The `get_inst()` function is used to retrieve a handle to the singlton map.

`static function this_t get_inst()`

Return a handle to the singleton map. The function is static so that it be called without an instance. The first time it is called an instance of the singleton map is created and a handle to it is stored locally. Subsequent calls to `get_inst()` will return the stored handle.

2.6 Queue

The queue is a first-in, first-out (FIFO) structure. `Put()` inserts an element into at the tail of the queue; `get()` retrieves an element from the head of the queue. The queue has an indeterminate size, you can keep putting elements into the queue until memory is exhausted.

Data Structure Interface

`virtual function void put(T t)`

Insert a new element at the tail of the queue.

`virtual function T get()`

Retrieve an element from the head of the queue. The element is removed from the queue.

`virtual function T peek()`

Retrieve an element from the head of the queue, but do not remove it. Repeated calls to `peek` without any intervening calls to `get()` will return the same element.

`virtual function bit is_empty()`

Returns one if there are no elements in the queue, returns one otherwise. The semantics are equivalent to evaluating the expression `(size() == 0)`.

Utility Interface

There is only one function in the utility interface, `clone()`. All other utility functions are in the base class.

```
function this_t clone()
```

2.7 Fixed Size Queue

Data Structure Interface

```
function new(int unsigned n = 1)
```

```
virtual function void set_max_size(int unsigned n)
```

```
virtual function int unsigned get_max_size()
```

```
virtual function void put(T t)
```

```
virtual function bit last_push_succeeded()
```

Utility Interface

```
function void copy(this_t q)
```

```
function this_t clone()
```

2.8 Stack

The stack is a last-in, first-out structure (LIFO). The last item added to the stack is the first one retrieved.

Data Structure Interface

`virtual function void push(T t)`

Put a new item at the top of the stack.

`virtual function T pop()`

Retieve the item at the top of the stack. If the stack is empty then the emtpy item is returned.

`virtual function bit is_empty()`

Return one if there are no elements in the stack, a zero otherwise. The semantics are equivalent to evaluating the expression `(size() == 0)`.

Utility Interface

There is only one function in the utility interface, `clone()`. All other utility functions are in the base class.

`function this_t clone()`

2.9 Deque

The queue is a two-way queue structure that is accessed at the front (head) and the back (tail).

Data Structure Interface

`virtual function T pop_front()`

Remove the item at the head of the list and return it. Return the empty item if the list is empty.

`virtual function T pop_back()`

Remove the item at the tail of the list and return it. Return the emtpy item if the list is empty.

`virtual function void push_front(T t)`

Insert a new item at the front of the list.

`virtual function void push_back(T t)`

Insert a new item at the tail of the list.

`virtual function void shuffle()`

Reorder the items in the list in a randomized order.

`virtual function void reverse()`

Reverse the order of the elements in the list.

Utility Interface

There is only one function in the utility interface, `clone()`. All other utility functions are in the base class.

```
function this_t clone()
```

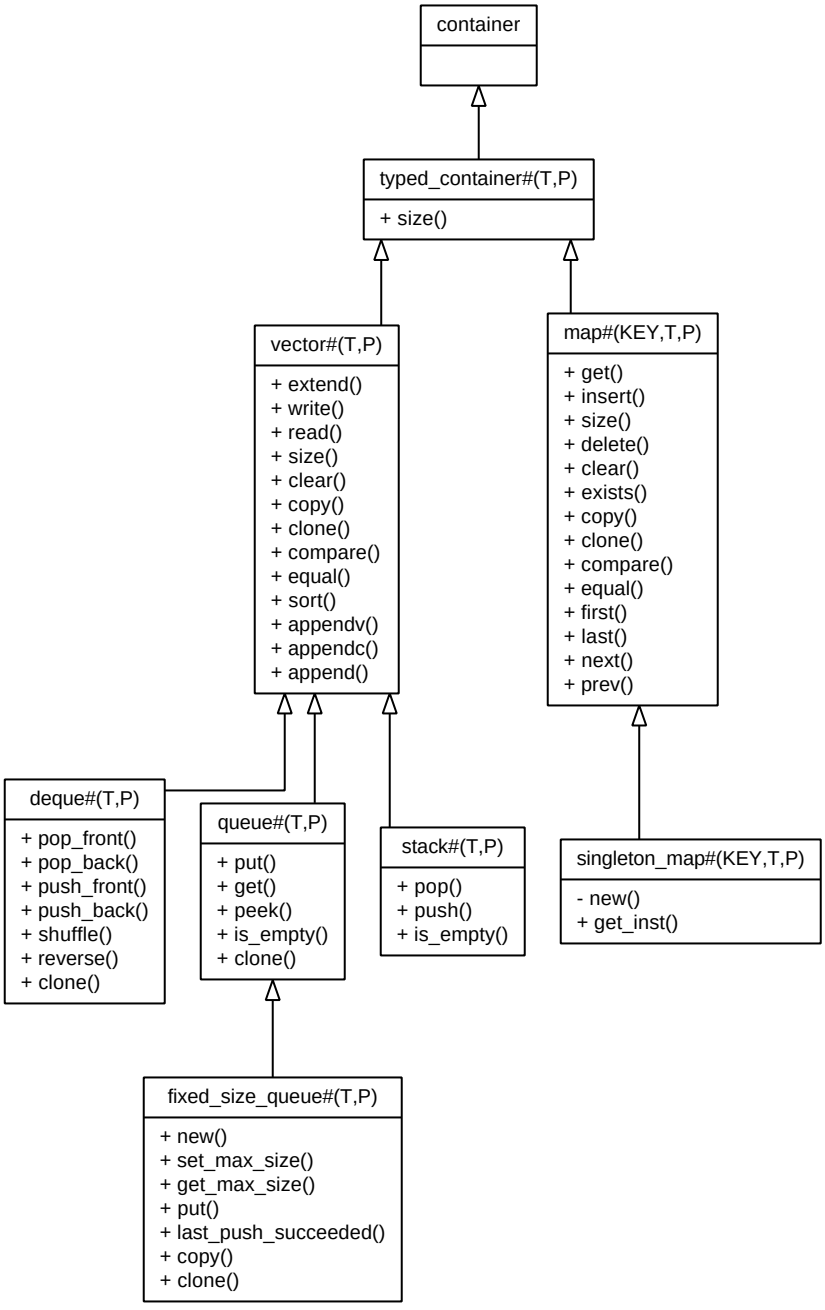

Figure 2.1: Container Inheritance

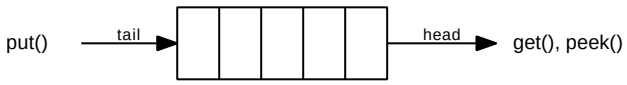

Figure 2.2: Queue Data Structure

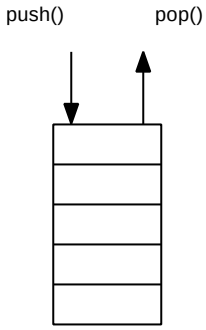

Figure 2.3: Stack Data Structure

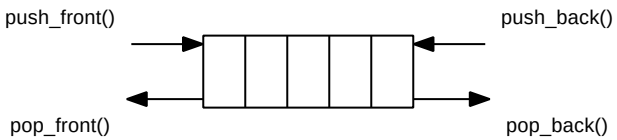

Figure 2.4: Deque Data Structure

Chapter 3

Iterators

A iterator is an object that binds to a container and provides means for traversing the contents of the container. The SVX library provdes several groups of iterators. *List iterators* are used for traversing vectors and containers derived from the `vector` class. *Map iterators* are for traversing maps and containers derived from the `map` class. *Permute iterators* are for traversing the permutations of items in a `vector` or containers derived from `vector`.

3.1 Iterator Interfaces

3.2 List Iterators

3.3 Map Iterators

3.4 Permute Iterators

SystemVerilog Extension Library (SVX)

This document was typeset entirely using open source text-based tools.

Typesetting was done using LaTeX, a markup program that is well-suited for building technical documents. More information about TeXand LaTeXcan be found on the TeX Users Group website (`tug.org`). LaTeXpackages can be found on the Comprehensive TeXArchive Network (CTAN) website (`www.ctan.org`).

Diagrams of networks, graphs, and trees were created using dot, a program that is part of the graphviz pacakge from AT&T Research. The graphviz website has all the details about the package, including dot (`www.graphviz.org`).