# Alternate Proposal on Class Declarations

Jay Lawrence

Cadence Design Systems, Inc.

3/06/2003

# 1 Introduction

During the SystemVerilog Enhancement Committee (sv-ec) meeting on February 10, 2003, Chapter 11 Classes of the SystemVerilog 3.1 Draft 2 LRM was discussed. During this meeting, there was a great deal of debate about the fact that variables of a class type can only be dynamically allocated but that there is no explicit indication that the object is dynamic. During this debate Jay Lawrence of Cadence proposed a motion that all objects of a class type have some additional indication that the object was dynamic. The intent of this motion was to allow for future extensions of SystemVerilog which would allow for dynamic allocation of structures and static allocation of classes thereby creating a regular type system.

There was debate of this topic, but since no detailed proposal was provided in advance of the meeting, many of the attendees did not fully understand the proposal and after debate, a vote was taken and the motion rejected. Following the debate some committee members expressed a desire for Cadence to document the proposal more fully so it could be more thoroughly understood. This document is that proposal and is provided to the sv-ec to provide the technical detail behind the motion and document the fact that Cadence believes a critical error is being made that will prevent simple orthogonal extensions of SystemVerilog data types in the future.

The issue at hand arises from that fact that the proposed SystemVerilog classes are always dynamically allocated objects and that all existing SystemVerilog variable data types are statically allocated. This is a historical difference created because SystemVerilog variables data types are derived from Superlog where the static nature of Verilog objects was preserved; whereas Classes are derived from Vera which uses a Java-like dynamic memory allocation paradigm. Each of these is a valid philosophy on data structures, but during the processing of these donations we believe that the committees have missed an opportunity to unify the concepts from the separate donations.

The remainder of this document first contains a proposal on how objects of the current class types can be declared. Following that a series of possible future enhancements to SystemVerilog are discussed and evaluated with and without the proposed changes. Finally, a detailed example of a fully parameterized linked list class is attempted (since I have access to neither a SystemVerilog or Vera toolset checking this model completely is impossible). The full specification is given with and without the proposed extensions including change bars between them to clarify the required change in coding style.

# 2 Proposal

The proposed change for the declaration of variables of a class type is that they require some indication of the dynamic nature of the object. In particular, the concept of a reference[1] to an object should be formalized in SystemVerilog and used for dynamic objects of all types, and as the semantic for pass-by-reference to tasks and functions.

A class is currently treated as a new kind of object which is implicitly dynamic in nature. Instead a class should be viewed as a new kind of data type that existing SystemVerilog variables can take on exactly like struct, integer, etc. This promotes a regular type system where dynamic structs and static classes naturally fall out through type orthogonality rather than introducing new syntax and semantics for each.

---

[1] The term reference is being used here. This could have just as easily been handle or descriptor. These are not equivalent to C++ references.

## 2.1 Reference Declarations

A reference is a descriptor that contains information about an object. A reference is not a raw pointer which indicates the address of an object. In particular, SystemVerilog references must be able to contain information on the fanout of the object referenced so that if multiple references refer to the same instance of a data object they can all detect and respond to changes in its value. Similarly, if an object is passed to a task by references, changes within the task must be visible outside the task immediately. Lastly all dynamically allocated object must have additional usage information stored with them to allow for garbage collection of memory.

The SystemVerilog type system should be extended to include the declaration of reference types. Initially due to time constraints on the standardization, these would only be defined for class objects, but could be extended to all data types in the future. A reference is declared by preceding the object name with the keyword **ref**[2].

For example:

```
module types;

        typedef class list_c;          // Forward declaration for class type
        typedef ref list_c list_r;      // Reference to a list_c type;

        class list_c ;
                logic [31:0] addr;
                logic [31:0] data;
                event        ready;
                list_r       next;   // A reference to another object of this type
        endclass
endmodule

module foo;

        types.list_r head;   // class use, this is a reference to an object
        ref types.list_c tail; // Alternative anonymous reference type

endmodule
```

## 2.2 Constructors

If a reference is declared to an object of a class type, the object is allocated and initialized using the class's constructor. This is unchanged from existing SystemVerilog proposal. To create a new object the constructor can be called with or without parameters as required by the constructor's parameter list.

```
        typedef class list_c;
        typedef ref list_c list_r;
        class list_c ;
                logic [31:0] addr;
                logic [31:0] data;
                event        ready;
                list_r       next;        // A reference to another object of this type

                function new (input logic [31:0] a = 32'b0, logic [31:0] d = 32'bx);
                        addr = a;
```

[2] There has been much debate about **ref** vs. **&** for this purpose. I now believe that in the general case '&' will cause problems due to its existence as both a unary operator and a reference indication, thus I've used ref.

```
            data = d;
            next = null;
        endfunction
    endclass
```

## 2.3 Reference Initialization

References are initialized to the value **null** by default. A reference object of a class type can contain an initialization expression which calls the constructor. This is unchanged from the existing proposal.

```
list_r head;          // initialized to null
list_r tail = null;   // explicitl initialization to null is allowed for the careful
list_r element = new;   // default values of parameters of new are applied
list_r element_2 = new(32'b1, 32'b0);  // explicit parameters to new
```

## 2.4 Dereferencing

The dereferencing of a reference simply uses the dot operator ('.'). This is unchanged from the existing proposal. The tools can always distinguish when it is necessary to perform an implicit dereference to get to the actual value of the object referenced.

```
logic [31:0] addr;
logic [31:0] data;

list_r head;

always
begin
    ....
    if (head != null)
    begin
        addr = head.addr;   // each of these dereference head
        data = head.data;
        head = head.next;
    end
end
```

## 2.5 Sensitivity

If an event control refers to a reference object, then the event control refers only to the value of the reference, not the object referred to[3]. In order to be sensitive to changes in a field of the object referred to, an event control on that field must be used. If a particular class requires that event controls be able to detect any change, then an event object should be embedded in the class which indicates a global change.

To wait on the reference:

```
if (head == null)
    @(head)   // waits until head becomes non-null
```

[3] Note, I don't feel strongly about this, but could not find an equivalent definition of sensitivity to class objects in the SystemVerilog LRM. We need to define this. The most interesting question is whether @(head) means any change in the referenced object, or any change in the reference itself.

To wait on a member of the class:

```
If (head != null)
        @(head.data)  // waits until the data field changes
```

To wait for a global change:

```
@(head.ready)    // Note this is an error if head is null
```

## 2.6 References to References

It shall be allowed to declare references to references. This is necessary to be able to write subroutines which manipulate reference values in the most general case. For example:

```
typedef ref list_r list_r_r;
```

There is no defined limitation to the level of reference indirection possible[4].

## 2.7 Declaring references to parameterized classes

If a reference refers to a parameterized class, then the object or typedef declaration must parameterize the underlying class.

## 2.8 Pass-By-Reference

Once the concept of a reference to an object as presented above exists, the concept of pass by reference should be made consistent with it because the reference has exactly the same semantics.

### 2.8.1 Passing static objects by reference - “ref” as an operator

In order to allow proper type checking of static objects passed by reference, a static object passed to a subprogram must use the **ref()** operator to pass the object.

```
typedef struct packed {
        logic [127:0] field1;
        logic [127:0] field2;
} bigstruct_s;

bigstruct_s bs;

task passbyref (inout ref bigstruct_s b)
….
endtask

passbyref ( ref(bs) );
```

The semantics of pass by reference would be unchanged from the current SystemVerilog semantics.

---

[4] This could be limited to 2 levels (a reference to a reference) and it would cover 95% of all the uses of references, but I see no reason to impose an arbitrary limitation.

### 2.8.2 Passing reference objects by reference

If a reference object is passed to a task or function with a parameter to be declared as a pass-by-reference parameter, then the object is simply passed unchanged, but the semantics of assignments to members of the object are exactly the same as for static objects (i.e. changes are seen immediately outside the task/function). The following example uses the previously defined list_r, list_r_r class and reference types and shows legal sequences of calls.

```
list_r lr;
list_r_r lrr;

task ref_task (list_r r);
task ref_ref_task (list_r_r rr);

ref_task(lr);                  // legal calls
ref_ref_task(lrr);
ref_ref_task( ref(lr) );
```

Note that no examples of a list_c type object or parameter are given. This is because this will be used in the future to declare a static object of the class type which is beyond the scope of SystemVerilog 3.1.

# 3 Future Enhancements

This section shows how some future enhancements to SystemVerilog would look if this proposal is adopted and how these future enhancements are made more complicated if this proposal is not adopted. The SystemVerilog technical committees will need to decide if any of these changes should be considered as a part of SystemVerilog 3.1 or for future revisions.

## *3.1 Static Objects of a Class Type*

In discussions around both Classes and the Program Block, experienced Vera users have proposed static instantiation of programs and classes. SystemVerilog is intended for adoption by Verilog users and they will expect to be able to create static hierarchies and class instances in testbenches and system-level models. Some modeling styles will bring an object into existence at time zero that will exist for the entire simulation. These are in effect design elements serving as testbench components representing the static structures that will eventually surround the device under test when it is integrated into a system. In addition, if classes are used in system-level models of real hardware, it may be more natural to have them static like all other variables in a hardware description. Arising from these discussions is a future enhancement to allow statically allocated classes.

A statically allocated class would simply be declared like any other SystemVerilog variable (continuing with the data types from the previous section.

```
list_c element;
```

This object would be allocated prior to the beginning of simulation at which time the constructor for the class would be implicitly called.

If a dynamic indication is added today, then the static class allocations would simply be the default for objects of a class type as they are for objects of all other data types. This would create a simple common declaration form for all objects which is simple to learn and remember.

If the dynamic indication is not provided now, then a statically allocated class object is going to need yet another keyword or indication that it is a static class object. This will add yet another keyword[5] and create confusion because this keyword is not required on static structs. For instance

```
static list_c element;
```

## 3.2 Dynamic Objects of a Variable Type

SystemVerilog has added many new data types to Verilog including structs. When these were added in SV 3.0 there was no provision made for dynamically allocating these structures and no concept of a reference or handle was added. This prohibits the modeling of dynamic data structures as is done in 'C' with structs to create such items as lists, queues, stacks, etc. One can argue that with 3.1 Classes this extension of structs will not be necessary in the future since a class could be used instead; however, this suggestion would force all uses wanting dynamic data structures to adopt object-oriented programming techniques. This would impose an Object Oriented (OO) methodology upon the Verilog design community that we believe is unnecessary, unwelcome and not warranted. There are certainly parts of the community that require support for OO techniques but the simple need for dynamic memory should not require that move.

In this proposal, we introduced the concept of an explicit reference. All data types could be extended to allow this syntax so that references to all SystemVerilog data types could be used.

```
typedef struct list_s;  // incomplete/forward declaration of struct type

// actual declaration of the structure
typedef struct {
        logic [31:0] addr;
        logic [31:0] data;
        ref list_s next;            // ref used to indicate this is a reference declaration
}  list_s;

ref list_s head;                    // declare a reference to the list head
```

Alternatively a typedef could be used to define the reference type as demonstrated for classes.

```
typedef struct list_s;
typedef ref list_s list_r;

list_r head;
```

An implicit constructor for all data types can be defined by the language which allocates an appropriate amount of memory, and initializes the values to the default values. Aggregate syntax could be used if explicit initialization was required.

If class objects are made to be dynamic without any such indication, then there is no visual indication when a type is used whether it is a class or a struct and therefore no indication whether it is dynamic or static.

```
module types;
        class list_c;
                logic [31:0] addr;
```

---

[5] We of course could reuse the currently added keyword **static** as shown here, but Cadence has objected to that keyword vehemently in the past and believes that the IEEE will remove it in the future as it rejected it in 1364-2001. Additionally, if it is not removed, how would you declare a statically allocated class object as a static variable in a task/function if the same keyword is used for both concepts?

```
        logic [31:0] data;
        list_t next;
    endclass

    struct list_s {
        logic [31:0] addr;
        logic [31:0] data;
        list_s &next;
    }
endmodule

module foo;
    …
    types.list_c lc;   // class use, this is a dynamic object
    types.list_s ls;   // structure use, this is a static object
    ….
endmodule
```

## 3.3 Future Unification of Struct and Class

Probably the most controversial but powerful future extension that should be considered in the future is the complete unification of structs and classes. If the committee sees the future need for dynamic structures and static classes, then the historical divisions between them should be removed. The only capabilities that exist for structs that are not a part of classes are:

- Classes can not be statically allocated
- Classes can not overlayed data (i.e. class members that are a union)
- Classes can not be declared as packed

By combining structs and classes multiple issues are solved:

- Overall number of keywords is reduced (duplicates are removed)
- Current semantics of static structures could be used to define semantics of statically allocated classes
- Current semantics of dynamic classes could be used to defined semantics of dynamic structures

As I believe this extension is well beyond the scope of SystemVerilog 3.1, no further development of the enhancement is provided here, however, if the declaration of static objects vs. reference objects is not adopted now then making them semantically equivalent (if not syntactically equivalent) in the future will be impossible.

# 4 Detailed Example

The following two sections contain an attempt at implementing a parameterized class specification for a linked list in SystemVerilog. The first section does it with the current definitions of classes; the second section does it with classes as modified by this proposal. I have used the tracking feature of Microsoft Word to highlight the changes required in going from the current proposal to the new one to highlight the changes. Each example is presented as if each subsection is a separate file, an attempt has been made to highlight keywords as is done in the LRM to make it simpler to read.

## 4.1 SystemVerilog Draft 3 LRM

This section provides a class specification for a parameterized linked list as currently specified in the Draft 3 LRM.

### 4.1.1 File “Orig_List.svh”

```
`ifndef LIST_C

`define LIST_C
typedef class List_c;

`endif
```

### 4.1.2 File: “Orig_ListElement.svh”

```
`ifndef LIST_ELEMENT_C

`define LIST_ELEMENT_C
typedef class ListElement_c;

`endif
```

### 4.1.3 File: “Orig_List.sv”

```
`include "Orig_ListElement.svh"
`include “Orig_List.svh”

class List_c;

        //
        // Declare a type parameter
        // (Note this parameter declaration syntax does not conform to the current spec
        //
        parameter type list_element_t = integer;

        //
        // head and tail of list
        //
        local ListElement_c #(list_element_t) head_, tail_;

        //
        // Create a new list
        //
        extern function List_c #(list_element_t) new;

        //
        // Push an element on the front of the list
        //
        extern function void push_front (ListElement_c #(list_element_type) element);

        //
        // Pop an element off the back of the list
        //
        extern function ListElement_c #(list_element_type) pop_back;

        //
        // Return the first element on the list, do not remove it
        //
        extern function ListElement_c #(list_element_type) front;
```

```
endclass;

function List_c::new;
begin
	head_ = null;
	tail_ = null;
	new = this;
endfunction

//
// Push an element on the front of the list
//
function void List_c::push_front (ListElement_c #(list_element_type) element);
begin

	ListElement_c #(list_element_type) old_head = head_;

	// This will be the new head of the list
	head_ = element;

	// If this is the only element it is also the tail
	if (!tail_)
		tail_ = element;

	//
	// This exploits the fact that we know how 'insert' is implemented
	// because we make sure head_ does not equal old_head so insert will put
	// it in the list and not recursively call push_front
	//
	element.insert(old_head, this);

	push_front = head_;

endfunction

//
// Pop an element off the back of the list
//
function ListElement_c #(list_element_type) List_c::pop_back;
begin

	ListElement_c #(list_element_type) old_tail = tail_;

	if (!tail_)
		pop_back = null;
	else
		tail_ = tail.prev();
		old_tail.detach();
		pop_back = old_tail;
	end

endfunction

//
// Return the first element on the list, do not remove it
//
```

```
function ListElement_c #(list_element_type) List_c::front;
begin
        front = head_;
end
```

### 4.1.4 File: “Orig_ListElement.sv”

```
// Include the header for the List and ListElement class
`include "Orig_List_c.svh"
`include "Orig_ListElement_c.svh"

class ListElement_c;

        //
        // Parameter for kind of list element
        // This syntax is not what is in the LRM for a parameter
        //
        parameter type list_element_t = integer;

        //
        // local data items to maintain an element
        //
        local list_element_t data_; // value of this list element
        local ListElement_c #(list_element_t) next_; // next list element
        local ListElement_c #(list_element_t) prev_; // previous list element
        local List_c #(list_element_t) list_; // list this element is in

        // Constructor
        extern function ListElement_c #(list_element_t) new (list_element_t element);

        //
        // Inserts an item before the position element
        // If the position argument is null, the plist argument can
        // indicate the list to which the item is added, if both are present they must agree
        //
        extern function ListElement_c insert (ListElement_c #(list_element_t) position,
                                              List_c #(list_element_t) plist = null);

        //
        // Detach an item from the list
        //
        extern function void detach;

        //
        // Return the data associated with a list element
        //
        extern function list_element_t data;

        //
        // Return the next and previous element in the list
        //
        extern List_Element_c #(list_element_t) next;
        extern List_Element_c #(list_element_t) prev;

        //
```

```
        // Return the list an element is in
        //
        extern List_c #(list_element_t) list;

endclass;

function ListElement_c #(list_element_t) ListElement_c::new (list_element_t element);
begin
        this.data_ = element;
        this.next_ = null;
        this.prev_ = null;
        this.list_ = null;

        // Is there an implicit 'new = this' at this point?
        new = this;
endfunction

//
// Inserts an element before the position element and return the new item
//
function ListElement_c #(list_element_t) ListElement_c::insert (
                                ListElement_c #(list_element_t) position,
                                List_c #(list_element_t) plist = null);
begin

        ListElement_c #(list_element_t) pos = position;
        List_c #(list_element_t) l = plist;

        if (!pos && !l)
                $error("Either position or list must be specified");

        else if (pos && l) begin
                if (position.list() != l)
                        $error("Position and list must agree on the list");
        end

        else if (pos && !l)
                l = pos.list();

        else if (!pos && l)
                pos = l.front();

        if (!pos) begin
                // This must be the first/only element
                this.list_ = l;
                this.next_ = null;
                this.prev_ = null;
        end
        else if (pos == l.front()) begin
                //
                // We are adding to the head of the list and if so call the list method
                // Note we are going to get called again immediately with head moved
                //
                l.push_front(this);
        end
        else begin
```

```
                //
                // Otherwise insert it in the list
                //
                this.next_ = pos;
                if (pos.prev_)
                begin
                        pos.prev_.next_ = this;
                        this.prev_ = pos.prev_;
                end
                pos.prev_ = this;
                this.list_ = l;
        end

        // Set the return value and return
        insert = this;
endfunction

//
// Detach an element from the list
//
function void detach;
begin

        ListElement_c #(element_type_t) p, n;

        p = item.prev_;
        n = item.next_;

        if (p)
                p.next_ = n;

        if (n)
                n.prev_ = p;

        item.next_ = null;
        item.prev_ = null;
        item.list_ = null;

endfunction

//
// Return the data associated with a list element
//
function list_element_t ListElement::data;
begin
        data = this.data_;
endfunction;

//
// Return the previous item
//
function ListElement_c ListElement::prev;
begin
        prev = this.prev_;
endfunction;
```

```
//
// Return the next item
//
function list_element_t ListElement::next;
begin
        next = this.next_;
endfunction;

//
// Return the list this item is in
//
function List_c #(list_element_t) ListElement::list;
begin
        list = this.list_;
endfunction;
```

## *4.2 SystemVerilog Draft 3 with Cadence Proposal*

This section provides a class specification for a parameterized linked list as modified with this proposal

### 4.2.1 File “Ref_List.svh”

```
`ifndef LIST_C

`define LIST_C
typedef class List_c;
typedef ref class List_c List_r;

`endif
```

### 4.2.2 File: “Ref_ListElement.svh”

```
`ifndef LIST_ELEMENT_C

`define LIST_ELEMENT_C
typedef class ListElement_c;
typedef  ref class ListElement_c ListElement_r;

`endif
```

### 4.2.3 File: “Ref_List.sv”

```
`include "Ref_ListElement.svh"

class List_c;

        //
        // Declare a type parameter
        // (Note this syntax does not conform to the current spec but is more Verilog-like)
        //
        parameter type list_element_t = integer;

        //
        // head and tail of list
        //
```

```
        local ListElement_c_r #(list_element_t) head_, tail_;

        //
        // Create a new list
        //
        extern function List_c #(list_element_t) new;

        //
        // Push an element on the front of the list
        //
        extern function void push_front (ListElement_rc #(list_element_type) element);

        //
        // Pop an element off the back of the list
        //
        extern function ListElement_rc #(list_element_type) pop_back;

        //
        // Return the first element on the list, do not remove it
        //
        extern function ListElement_rc #(list_element_type) front;

endclass;

function List_c::new;
begin
        head_ = null;
        tail_ = null;
        new = this;
endfunction

//
// Push an element on the front of the list
//
function void List_c::push_front (ListElement_rc #(list_element_type) element);
begin

        ListElement_rc #(list_element_type) old_head = head_;

        // This will be the new head of the list
        head_ = element;

        // If this is the only element it is also the tail
        if (!tail_)
                tail_ = element;

        //
        // This exploits the fact that we know how 'insert' is implemented
        // because we make sure head_ does not equal old_head so insert will put
        // it in the list and not recursively call push_front
        //
        element.insert(old_head, this);

        push_front = head_;

endfunction
```

```
//
// Pop an element off the back of the list
//
function ListElement_rc #(list_element_type) List_c::pop_back;
begin

        ListElement_rc #(list_element_type) old_tail = tail_;

        if (!tail_)
                pop_back = null;
        else
                tail_ = tail.prev();
                old_tail.detach();
                pop_back = old_tail;
        end

endfunction

//
// Return the first element on the list, do not remove it
//
function ListElement_rc #(list_element_type) List_c::front;
begin
        front = head_;
end
```

### 4.2.4 File: “Ref_ListElement.sv”

```
// Include the header for the List and ListElement class
`include "Orig_List_c.svh"
`include "Orig_ListElement_c.svh"

class ListElement_c;

        //
        // Parameter for kind of list element
        // This syntax is not what is in the LRM for a parameter
        //
        parameter type list_element_t = integer;

        //
        // local data items to maintain an element
        //
        local list_element_t data_; // value of this list element
        local ListElement_rc #(list_element_t) next_; // next list element
        local ListElement_rc #(list_element_t) prev_; // previous list element
        local List_rc #(list_element_t) list_; // list this element is in

        // Constructor
        extern function ListElement_rc #(list_element_t) new (list_element_t element);

        //
        // Inserts an item before the position element
        // If the position argument is null, the plist argument can
```

```
        // indicate the list to which the item is added, if both are present they must agree
        //
        extern function ListElement_rc insert (ListElement_rc #(list_element_t) position,
                                               List_rc #(list_element_t) plist = null);

        //
        // Detach an item from the list
        //
        extern function void detach;

        //
        // Return the data associated with a list element
        //
        extern function list_element_t data;

        //
        // Return the next and previous element in the list
        //
        extern List_Element_rc #(list_element_t) next;
        extern List_Element_rc #(list_element_t) prev;

        //
        // Return the list an element is in
        //
        extern List_rc #(list_element_t) list;

endclass;

function ListElement_rc #(list_element_t) ListElement_c::new (list_element_t element);
begin
        this.data_ = element;
        this.next_ = null;
        this.prev_ = null;
        this.list_ = null;

        // Is there an implicit 'new = this' at this point?
        new = this;
endfunction

//
// Inserts an element before the position element and return the new item
//
function ListElement_rc #(list_element_t) ListElement_c::insert (
                                ListElement_rc #(list_element_t) position,
                                List_rc #(list_element_t) plist = null);
begin

        ListElement_rc #(list_element_t) pos = position;
        List_rc #(list_element_t) l = plist;

        if (!pos && !l)
                $error("Either position or list must be specified");

        else if (pos && l) begin
                if (position.list() != l)
                        $error("Position and list must agree on the list");
```

```
        end

        else if (pos && !l)
                l = pos.list();

        else if (!pos && l)
                pos = l.front();

        if (!pos) begin
                // This must be the first/only element
                this.list_ = l;
                this.next_ = null;
                this.prev_ = null;
        end
        else if (pos == l.front()) begin
                //
                // We are adding to the head of the list and if so call the list method
                // Note we are going to get called again immediately with head moved
                //
                l.push_front(this);
        end
        else begin
                //
                // Otherwise insert it in the list
                //
                this.next_ = pos;
                if (pos.prev_)
                begin
                        pos.prev_.next_ = this;
                        this.prev_ = pos.prev_;
                end
                pos.prev_ = this;
                this.list_ = l;
        end

        // Set the return value and return
        insert = this;
endfunction

//
// Detach an element from the list
//
function void detach;
begin

        ListElement_rc #(element_type_t) p, n;

        p = item.prev_;
        n = item.next_;

        if (p)
                p.next_ = n;

        if (n)
                n.prev_ = p;
```

```
        item.next_ = null;
        item.prev_ = null;
        item.list_ = null;

endfunction

//
// Return the data associated with a list element
//
function list_element_t ListElement::data;
begin
        data = this.data_;
endfunction;

//
// Return the previous item
//
function ListElement_rc  ListElement::prev;
begin
        prev = this.prev_;
endfunction;

//
// Return the next item
//
function ListElement_rc ListElement::next;
begin
        next = this.next_;
endfunction;

//
// Return the list this item is in
//
function List_rc #(list_element_t) ListElement::list;
begin
        list = this.list_;
endfunction;
```

# 5 Conclusion

The addition of classes to provide Object Oriented programming techniques is a powerful addition to SystemVerilog 3.1, however the default dynamic nature of class declarations as currently proposed causes problems with many future issues for extensions to SystemVerilog. The treatment of classes as a data type applied to the current SystemVerilog variables rather than as a new kind of object leads to a regular type system more easily defined and extended. The simple addition today of an explicit indication that the declared object is a reference to a dynamic object will permit the future additions of:

- Other dynamic variable types
- Statically allocated objects of a class type
- The unification of dynamic object references and pass-by-reference
- A The future reconciliation of structs and classes into a single coherent composite object type

If we do not make this simple change at this time, all of the above future extensions will become much more difficult if not impossible.