Analysis of the "Alternate Proposal on Class Declaration"

Arturo Salz
Synopsys, Inc
3/13/2003

In order to analyze the technical merits of the "Alternate Proposal on Class Declarations", we first review the goals set forth by the proposal's introduction. The introduction makes a set of remarks (reproduced below in blue) that loosely constitute the requirements.

"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."

→ Requirement 1: Explicit indication that a class object is 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."

→ Requirement 2: Allow dynamic allocation of structures.
→ Requirement 3: Allow static allocation of classes.

"…Cadence believes a critical error is being made that will prevent simple orthogonal extensions of System-Verilog data types in the future."

→ Requirement 4: Enable simple orthogonal extensions of SystemVerilog data types in the future.

Later in the document some of these extensions are identified as:
1. Static objects of a class type
2. Dynamic objects of a variable type
3. Unification of struct, union, and class

It's clear from the requirements that the proposal does not attempt to solve a deficiency in the language, nor does it establish that a problem exists that couldn't be solved with the existing class paradigm. Moreover, the proposal doesn't add any additional semantic power to the language. As we will show later, everything that can be done with the alternative proposal can be done with the current language, but with a lot less typing and confusion. The proposal's goal is simply an academic exercise that seeks to create "a regular type system" that will enable "simple orthogonal extensions in the future". Whether one agrees with these goals is irrelevant, what really matters is what additional semantic power the proposal provides to end users, and at what cost. It is our contention that the proposal adds nothing in terms of semantic capabilities, and the immediate cost is a much more complex and verbose language.

A secondary cost but an important consideration of some of the "simple orthogonal enhancements", such as the dynamic objects of other variable types, is the potential performance degradation due to the existence of aliases throughout the design. However, that is a topic for another discussion since the current class mechanism neither enables nor prohibits that extension.

The introduction justifies the proposal with the following statements:

"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."

Neither of these two is completely accurate. First, there are several new data types in SystemVerilog-3.1 that are always dynamically allocated and are not classes. These are string, associative arrays, dynamic arrays, some uses of event variables, semaphores, mailboxes, and all the data associated with background processes (fork ..join_none). Furthermore, neither of these data-types uses an explicit indication that they are dynamic, nor is one warranted.

Second, the statement that "Superlog variables are of a static nature" is incorrect. The synthesizable subset of Superlog that was donated to Accellera obviously consists of static variables, but Superlog does allow for dynamically allocated data as well, it simply isn't synthesizable. And, Vera does indeed borrow heavily from Java's class paradigm, nevertheless it also provides for statically allocated variables. In fact, Vera has almost the same static variables as Verilog-2001. Notwithstanding, both Vera and Java have regular type systems. What these two languages do not have is a synthesizable language subset, and therein lies the crux of the issue. To what extent must the type system maintain different aggregate types in order to support both synthesizable structs and easy-to-use dynamically allocated classes? Is the goal of unifying struct and class into a regular type system just a delusion or a real, attainable goal?

Synopsys believes that the class mechanism as currently specified provides a powerful addition to SystemVerilog-3.1 that has proven to be extremely useful in the creation of test-benches. This mechanism is uncomplicated for the most common use of classes: dynamically allocated objects that are passed by reference (this is true even for C, in which structs are allocated dynamically, and passed-by-reference almost exclusively). We also understand that polymorphic classes are not synthesizable (nor are they likely to be anytime soon), whereas struct's are being used today with synthesizable designs. Thus, we believe that the user's needs are best served by keeping these two constructs separate. Unification of aggregate types is definitely a delusion.

The proposal goes on to state in section 3 that:

"experienced Vera users have proposed static instantiation of programs and classe … Some modeling styles will bring an object into existence at time zero that will exist for the entire simulation…. 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."

These statements perpetuate two misconceptions. First, that experienced Vera users would look favorably on the proposal. However, nothing could be further from the truth (and we know Vera users, experienced or otherwise). If there is one thing on which most Vera users will agree is how straightforward and easy to use Vera's classes are. They have none of the complications associated with pointers, address-of, and dereferencing operators that permeate C/C++, not to mention the crashes due to dangling references, prematurely de-allocated objects, etc.

The other misconception is that the existing language lacks the ability to declare a class that comes into existence at time zero prior to the start of simulation and will exist for the entire simulation. This is already supported in a clean and straightforward manner:

```
const Foo obj = new;     // Foo is a class
```

The declaration above ---if specified at a module (interface or program) level --- specifies that the object `obj` is to be created once before the start of simulation, and its handle, `obj,` will exist and remain constant for the entire simulation.

Furthermore, the existing class mechanism allows multiple parameterized classes to be declared with the exact same semantics. For example:

```
const Cartoon mouse = new( "Mickey" );
const Cartoon duck = new( "Donald" );
```

Contrast this with the alternate proposal for declaring static classes, which states that:

```
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."

Since the constructor is implicitly called, users loose the ability to parameterize the static class, and would be forced to use a dynamic class. This is a clear loss in functionality. Note that the only real, semantic difference between the two mechanisms is the segment from which the object's memory is allocated (the text segment or the heap segment). This distinction is probably beyond most Verilog users and a highly irrelevant one. Furthermore, if the class constructor includes calls to new (for example a static list) then even that distinction wouldn't be true.

Not only that, but the proposal attempts to make this weaker and less powerful mechanism the default:

"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."

Nonsense! The most common situation is dynamic classes that are passed by reference (i.e., handles) and that's what merits optimization. Moreover, this creates two different mechanisms for declaring classes; how can this be simpler or easier to learn? A single mechanism that optimizes the most common usage is clearly a better choice.

"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 and create confusion because this keyword is not required on static structs". For instance

```
static list_c element;
```

So, this is all an attempt to perhaps save a keyword when implementing a feature that is hardly ever going to be used (and we know this from our experience with C/C++). Moreover, we believe this feature will never need to be added since the "static" semantics are already supported, and in a more powerful way than the proposed static classes can ever achieve.

In section 3.2, the proposal states another common misconception.

"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."

The notion that classes impose a strictly object-oriented methodology is simply false. If users choose not to use an OO methodology then they simply need not declare any methods in their classes (nor use inheritance). Then, classes behave exactly like a dynamic C struct (albeit the allocator is called new and not malloc). Plenty of Vera users are uncomfortable with OO techniques, yet they have no problem with using classes as they are.

While the ability to add simpler (non-aggregate) dynamic date-types might be a worthwhile future extension, it is a completely orthogonal issue to the existing class declaration. At best, this is a philosophical issue regarding the regularity and orthogonality of the language. However, the performance implications must be carefully considered before adding a feature that potentially aliases every single bit in the design, thereby precluding most global optimizations that has taken vendors over a decade to devise and implement.

Proposal Review

What the alternate proposal essentially entails is that users will have two schemes for class instance creation:

1. Static object, like top-level instances in C++.
2. Dynamic objects accessed though a *reference*.

In the proposal, static objects are the default, and dynamic objects are made more cumbersome by the requirement of an additional keyword: **ref**. We contend that the dynamic case is more common and should thus be given more importance.

The proposal talks about references but in reality, it introduces explicit pointers (this is precisely what the proposals references are), with an explicit distinction between pointer and object access (as evidenced by the wait-on-reference example). This choice introduces additional complexity and possibly confusion, since users need to make the distinction between a pointer-to-object and an object when declaring tasks and functions. Since the most common use is pass by reference then an additional keyword per parameter is necessary, or an additional typedef to declare the pointer type (as in the code examples).

In the current language (and in Vera), the pointer is implicit, and there is no explicit distinction between reference and object. It is Synopsys' view that this leads to more focus on algorithms and less emphasis on details, since users are not concerned with thinking about explicit pointers or objects.

The alternate proposal provides for the equivalent of C/C++'s address-of operator, but it does not provide a dereference operator, (C/C++s *). This leads to confusion and ambiguity when more than one level of indirection is needed, and the confusion led to the scrapping of an earlier proposal that allowed arbitrary levels of indirection. However, if a dereference operator is provided the syntax will become so much more compex.

In the discussion on parameterized classes, the proposal mentions that the reference type has to be parameterized as well, but the proposal doesn't show that. Is the parameter implied? That raises the question as to whether the issue is properly understood.

Although we have already discussed why unification of struct and class is neither desirable nor warranted, we take strong issue with the assertion that the language needs to unify not just struct and class, but union as well. If this is allowed to happen, then the strongly typed class system, which has been carefully crafted as to avoid the most common memory pitfalls that befall C/C++, will be utterly destroyed. Users will be able to (perhaps unwittingly) cast one pointer to another and either get bogus results or crash with some memory violation, and, the tool will be unable to provide any assistance.

Conclusion

The addition of classes is a powerful addition to SystemVerilog 3.1. The default dynamic nature of class declarations optimizes the most common use model and leads to a simpler and easier to use language. The type system regularity espoused by the alternate proposal comes at the expense of a much more verbose and complex language.

Some of the extensions contemplated by the proposal are either ill-advised or down right impossible (at least not possible with 21st century technology). The introduction of an explicit distinction between pointers and objects creates much confusion and complexity that we believe is not worthwhile in order to achieve the blending of class objects with existing static data types. The unification is achieved, but the downside is that the most common usage becomes much more complex and hard to use.

We have also shown that the current language provides for the same semantics as static classes, but using a simpler, more general mechanism that also allows class instances to be parameterized. We also show that the existing language is at least as powerful as the alternate proposal, but with less confusion and complexity.

Finally, as mentioned at the start, the alternate proposal does not attempt to solve a semantic deficiency in the language, nor does it state a problem that cannot be solved with the current class mechanism. It merely attempts to unify several constructs that we feel are best as they are, otherwise ease-of use, performance, and robustness will all be adversely affected.