Introduction to SLX 2.0

1.0 Philosophy

The most important aspect of SLX is its *extensibility*. SLX provides a collection of well-designed and efficiently implemented core simulation primitives, *and* it provides mechanisms for extending core features into a wide diversity of unique, high-end applications. Key ingredients to SLX's success include its expressiveness and its ability to quickly and clearly expose errors, both in model construction, i.e., at compile-time, and during model execution.

Much of SLX's expressiveness is due to its object-based paradigms. Wolverine has always taken the position that SLX is object-based, but not truly object-oriented. With the advent of SLX 2.0, that is changing to a considerable degree. SLX 2.0 incorporates new (to SLX) object-oriented constructs that have been taken from a variety of sources, including Java, C++, and C#. The new features reflect many discussions with SLX users and incorporate many of their suggestions. As always, Wolverine has attempted to embrace technology without "going off the deep end." We've tried to keep the complexity down to a minimum that offers great bang for the buck.

2.0 Subclasses and Inheritance

The most important new capability in SLX 2.0 is the ability to define subclasses. Throughout this document, we'll use the following hierarchy of classes in our discussion and examples:

vehicle
- car
- truck
- tractor
  - JDtractor
  - Ctractor

car, truck, and tractor are specialized kinds of vehicles. JDtractor and Ctractor are specialized kinds of tractors.

In SLX 1, the only mechanism for combining objects is *composition*. The lowest-level objects are built first, and higher-level objects can include instances of the lower-level objects. In SLX 1, a portion of the hierarchy shown above could be implemented as follows:

```
class vehicle
    {
    vehicle attributes…
    };

class tractor
    {
    tractor attributes…

    vehicle     BaseVehicle;    // incorporate an instance of a vehicle object
    };

class JDtractor
    {
    JDtractor attributes…

    Tractor     BaseTractor;    // incorporate an instance of a Tractor object
    };
```

If "jdt" is a pointer to a JDtractor, and "vprop" is a member of the vehicle class, the "vprop" accessed by "jdt" is specified as

```
jdt -> BaseTractor.Basevehicle.vprop
```

While this approach is workable, it's very cumbersome. If any changes are made in the hierarchy, such as adding an intermediate level, updating all references to class members is a tedious, error-prone process.

Furthermore, suppose you wanted to have a set of vehicles; i.e., a set comprised of cars, trucks, tractors, JDtractors, and Ctractors. In SLX 1, you only choice would be to employ a universal set (set(*)). While extremely useful, universal sets carry the risk that objects of the "wrong" types can be placed in them. For example, if you're building a transportation model, and "Traffic-Light" is an object class, you could inadvertently place TrafficLight objects into sets that were intended to contain only vehicles.

The use of subclasses, in which a subclass *inherits* its parent's attributes, makes life much easier:

> Note that SLX 2.0 supports *single* inheritance only. That is, a given class can be a subclass of at most one parent class.

```
class vehicle
    {
    vehicle attributes…
    };

class tractor subclass(vehicle)        // tractors inherit vehicle properties
    {
    tractor attributes…
    };

class JDtractor subclass(tractor)      // JDtractors inherit tractor properties
    {
    JDtractor attributes…
    };
```

You can think of the "subclass" clause as a transitive "is a" relationship; i.e., a JDtractor *is a* tractor *is a* vehicle. Accessing the "vprop" for a given "jdt" now takes the form

```
jdt -> vprop
```

Parent classes can be used in set definitions:

```
set(vehicle) vset;
```

Such a set could be used to contain vehicle objects or any objects whose classes are *derived* from the vehicle class. Thus, vset can contain vehicles, cars, trucks, tractors, JDtractors, and Ctractors, but not TrafficLight objects.

Classes and subclasses in a hierarchy can be parameterized (take arguments), but severe restrictions apply to the forms of arguments that can be used. Consider the following example:

```
class vehicle(string(50) initial_vname))
    {
    };
```

```
class truck(string(50) vehicle_name, double tonnage)

    subclass(vehicle(vehicle_name))
      {
      };

  t = new truck("My Pickup", 5.0);
```

The first argument of truck is passed through to vehicle. In general, pass-through arguments must be arguments of the current class, constants, or global variables. Procedure calls / method invocations are explicitly disallowed. The reason for the latter limitation is that at the point at which arguments to a parent class are supplied, the initialization status of the parent class and the child class passing it arguments is somewhat tenuous. On the one hand, you could argue that the parent class should be initialized first (including calling its *initial* property, if any). This makes sense, since by definition, the parent class cannot include references to attributes of its children. On the other hand, if procedure calls / method invocations were allowed as arguments to the parent class, the possibility exists that the procedure could refer to elements of the child class. This would require initializing the child class *before* initializing the parent class. While a set of rules could be worked out to allow this sort of thing under very tightly controlled circumstances, enforcement of any such rules by SLX would be very complicated, and remembering the rules would be very difficult for users. Thus, the decision was made to impose the severe limitations.

3.0 Pointer Promotion and Demotion

Assume that we have the following collection of pointers:

```
pointer(vehicle)     v;
pointer(car) c;
pointer(tractor)     t;
pointer(JDtractor)   jdt;
pointer(Ctractor)    ct;
```

Assignments such as "v = c" constitute pointer *demotion*. No error checking is required for such assignments. This is true for the current example, because a car *is a* vehicle.

Conversely, assignments such as "t = v" constitute pointer *promotion* and require run-time validation. In this example, v must point to a tractor, a JDtractor, or a Ctractor, or be NULL; otherwise a run-time error occurs.

Since SLX sets are collections of pointers to objects, promotion and demotion apply analogously to sets of objects.

4.0 Enhancements to SLX Methods

In SLX 1, there was no "method" keyword. SLX *procedures* contained within classes were considered to be methods. In SLX 2, the preferred keyword is "method", rather than "procedure," although the latter form is still accepted, since so many existing SLX programs already use this notation.

4.1 Inherited, Overridable Methods

In SLX 2.0, the methods of a parent class are inherited by its descendant classes. For example, consider the following expanded definition of the vehicle class:

```
class vehicle
    {
    vehicle attributes…

    method GetVehiclePosition(out double x, out double y)
        {
        …
        }
    };
```

Since a JDtractor is a vehicle, we can use

```
jdt -> GetVehiclePosition(x, y)
```

to determine the current position of a JTtractor object.

A method of a parent class can be overridden in a descendant class, but only if the method is declared to be *overridable* in the parent class and declared to be an *override* in the descendant class:

```
class vehicle
    {
    overridable method GetVehiclePosition(out double x, out double y)
        {
        …
        }
    …
    };

class tractor subclass(vehicle)
    {
    override method GetVehiclePosition(out double x, out double y)
    …
    };
```

The required pairing of "overridable" and "override" protects you against unintentional overrides of methods.

Finally, note that a method override can itself be overridden. Currently, SLX assumes that once a method has been overridden, further overrides in descendant classes are allowed without requiring the "overridable" designation. In other words, you don't need to specify "overridable override".

4.2 Abstract and Concrete Methods

A parent class can contain *abstract* methods, e.g.,

```
class vehicle
    {
    …
    abstract method GetVehiclePosition(out double x, out double y);
    };
```

An abstract method is a method that must be supplied in any subclass of the parent class. Note that the body of an abstract method is empty. The declaration of an abstract method can be regarded as a contract unilaterally enforced by the parent class: "if you want to be a subclass of me, you must provide concrete instances of my abstract methods."

Each child class must contain a *concrete* method instance such as the following:

```
class tractor subclass(vehicle)
    {
    concrete method GetVehiclePosition(out double x, out double y)
        {
        …
        }
    …
    };
```

The required pairing of "abstract" and "concrete" protects you against misuse of methods. Any method described as concrete must have a corresponding abstract method in its parent class.

Note that a subclass may pass on to its subclasses the requirement to provide concrete instances of its parent's abstract methods. Consider the following example:

```
class parent
    {
    abstract method M();
    …
    };

class child subclass(parent)
    {
    abstract method M();      // pass-through to grandchild
    …
    };

class grandchild subclass(child)
    {
    concrete method M()
        {
        …
        }
    };
```

Note that in the "child" subclass, for consistency, method M should have been defined as "abstract concrete" ("concrete abstract"?). Why? Method M is concrete in the sense that it is a required instance of parent's abstract method M, and it's abstract in the sense that the requirement to instantiate it is passed on to the grandchild class. Although logically consistent, "abstract concrete" is quite ugly. Accordingly, SLX allows you to specify "abstract" only, if you'd prefer. Allowing this notational shortening is analogous to not requiring "overridable override."

Concrete methods can be overridden if an "overridable" prefix is supplied:, e.g.,

```
overridable concrete method GetVehiclePosition(out double x, out double y)
```

4.2 Sealed Methods

A *sealed* method is a method that cannot be further overridden, i.e.,

```
sealed concrete method GetVehiclePosition(out double x, out double y)
```

The "sealed" prefix can be applied to classes and modules, thereby causing all methods in a module/class to be sealed by default.

5.0 Interfaces

SLX *interfaces* are collections of abstract methods and abstract variables. Note that the ability to include abstract variable declarations in an interface is typically not found in other languages that include interfaces.

```
interface HasPosition
    {
    abstract double  TimeOfLastUpdate;
    abstract method GetPosition(out double x, out double y);
    abstract method SetPosition(in double x, in double y);
    };
```

Any given class can be declared to *implement* a collection of one or more interfaces, e.g.,

```
class vehicle implements(HasPosition)
    {
    …
    concrete double  TimeOfLastUpdate;
    concrete method Getposition(…)
    concrete method Setposition(…)
    };
```

An interface can be thought of as a form of contract. Any class that implements an interface must supply concrete instantiations of all the abstract declarations in the interface.

An interface is very similar to a class, but contains only abstract definitions and is used as a superclass. Consider the following example:

```
class HasPosition
        {
        double          TimeOfLastUpdate;
        abstract method GetPosition(out double x, out double y);
        abstract method SetPosition(in double x, in double y);
        };

    class vehicle subclass(HasPosition)
        {
        …
        concrete method Getposition(…)
        concrete method Setposition(…)
        };
```

The two examples shown are functionally equivalent. You might question what value interfaces offer. The answer is that interfaces are a partial substitute for multiple inheritance, which SLX 2.0 does not support. In SLX 2.0, a class can be a subclass of at most one other class; however a class can implement more than one interface, e.g.,

```
class vehicle implements(HasPosition, HasMotion)
```

In addition, interfaces can be used in conjunction with class hierarchies. For example, any class in our vehicle-car-truck-tractor-JDtractor-Ctractor hierarchy can be declared to implement one or more interfaces, and interface implementation is inheritable.

One of the most commonplace uses of interfaces is to declare procedure/method arguments as pointers to an interface, e.g.,

```
procedure UpdatePosition(pointer(HasPosition) hp)
```

Arguments to the above procedure can be pointers to objects of any classes that implement the HasPosition interface.

Pointers to interfaces can be copied, with the expected rules for promotion and demotion:

```
pointer(HasPosition)    hp;
pointer(vehicle)        v;
pointer(truck)          t;
…
hp = v;     // always OK, because vehicle implements HasPosition
hp = t;     // always OK, because truck is a vehicle
t = hp;     // requires validation, since hp can point to any class that implements
              HasPosition
```

Suppose that rather than having class vehicle implement HasPosition, only class tractor implemented HasPosition. If this were the case, the following rules would apply:

```
hp = v;     // OK if v points to a tractor, JDtractor, or Ctractor or is NULL
hp = t;     // always OK
t = hp;     // requires validation, since hp can point to any class that implements
            // HasPosition
```

Note that since interfaces are totally abstract constructs, you cannot create instances of them; e.g.

```
hp = new HasPosition;
```

is illegal.

It is possible to have a set of objects from classes that support a common interface, e.g.,

```
set(HasPosition)      ObjectsWithPositions;
```

However, interface-based sets cannot be ranked. This is because all ranked sets require specification of ranking attributes (class members), and because interfaces are collections of abstract methods *only*, interfaces by definition cannot contain attributes to be used for ranking.

6.0 Protected Class Members

In SLX 1, the *visibility* of class members (public or private) and *access rights* (read_only or write_only) were enforced on a module basis. For example, private class members were visible inside the module in which they were defined, but invisible elsewhere.

SLX 2.0 adds a new keyword, "protected," that is enforced on a *class hierarchy basis*. Members declared to be protected in a given class are visible throughout that class and its methods, and visible throughout any subclasses (and their methods) derived from the original class. The "protected" keyword can also be applied on a class-wide basis. If this is done, individual declarations within the class can override the default protected status by using an "unprotected" prefix. Consider the following examples:

```
class widget
    {
    protected int      counter;
    };

protected class controlled_access
    {
    int                counter1;
    unprotected int    counter2;
    };
```

7.0 Interaction of Specifications

In previous sections, we've discussed a number of keywords that apply to classes, class members, and methods. Supplying more than one keyword can lead to conflicting specifications. For example, declaring a method to be both abstract and sealed makes no sense, because an abstract method must be defined in a class's subclass, and a sealed method cannot be further modified. For the most part, when conflicting specifications are given, the most recent (rightmost) specifications are used without complaint. For some truly egregious conflicts, SLX may issue warning or error messages. "Silent" conflict resolution rules are shown in the following table:

Keyword	Implied Keywords	Canceled Keywords
abstract		sealed, overridable
concrete		
overridable		abstract, sealed
override		
protected	private	public, overridable
unprotected		protected
sealed		overridable, abstract

8.0 Developing Your Own Style

The features added to SLX 2.0 enlarge the set of tools for organizing and managing large projects. Wolverine goes to great pains to avoid dictating style. Nevertheless, we offer the following suggestions for effective use of the new features:

A. Well-designed class hierarchies greatly facilitate software reuse. For a given application, over time, changes to classes that are close to the root of a class hierarchy should become less and less frequent, and introduction of new classes that are specialized versions of previously implemented classes should be easier.

B. Try to avoid the use of pointer(*) and set(*). While these constructs were necessary in SLX 1, because "anything goes" in their use, SLX is very limited in the kinds of compile-time and run-time checking it can do. In contrast, using pointers to the roots of class hierarchies and sets of root class objects offers much of the flexibility of pointer(*) and set(*), but adds the protection that accrues from well-defined rule for promotion and demotion.

C. Carefully weigh the tradeoffs between using class hierarchies and interfaces. In lots of cases, you'll want to use both. Interfaces are very effective for defining procedures that perform common operations across a diverse collection of object classes, while class hierarchies are great for closely related families of object classes.

D. If you're part of a development team, carefully weigh the issues of who has access to what, and how. For example, protected class members can be used for data that only class implementers should have access to. For the ultimate protection, you can collect into a file modules containing code that is intended for use by non-developers. Compile this file and create an RTS file from it. The RTS file can then be incorporated into code written by others by using "import" statements. When this is done, the source code from which the imported file was created is invisible.

E. Beware of the cost of over-modularizing code. The overuse of inheritance and interfaces can result in lots of very short methods that retrieve stored values or perform simple calculations on them. For example, one could define methods for converting degrees to radians and vice versa. In SLX, the overhead of procedure and method calls is higher than it is in most languages, which is in part due to SLX's policy of initializing all variables with known values and in part due to SLX's use of a disjoint run-time stack (necessary for supporting suspension and resumption of puck movement). Thus, the run-time cost of overly-modularized code can be undesirably high. An interesting alternative is to use macros. Of course, one must weigh the benefits of providing more controlled access to data versus run-time efficiency.

9.0 Sample Programs

The following sample programs are being distributed with the prototype version of SLX 2.0:

File	Contents
inheritance.slx	Illustrates the vehicle class hierarchy used in this paper. By carefully stepping through the program one line at a time, you'll be able to see how everything fits together. The file contains many lines that are commented out that would generate compile-time or run-time errors if not commented out.
interfaces.slx	A basic tour of interfaces.
xfacevar2.slx	Shows the use of abstract variable declarations in interfaces.
protected.slx	Shows how to protect access to class members.
JOHstate.slx	A small, but complete example of inheritance.