6 The Modeling Language NSLM

In this chapter we describe the NSLM modeling language. NSLM is a high-level programming language designed to support the construction of model architectures in NSL. For efficiency and extensibility reasons, the NSLM language is translated into either C++ or Java, depending on the chosen environment. While NSLM is a *self-contained* programming language supporting a complete set of types and expressions—the user may take advantage of the full power of C++ and Java when necessary. Yet, we strongly recommend avoiding as much as possible writing "direct" Java or C++ code but try to follow NSLM modeling philosophy and expressions as much as possible. This will result in more consistent and extensible code. In general terms, NSLM is actually a superset of either language in that it provides a set of types and expressions common to both languages together with a library of classes useful in constructing and simulating models in NSL. NSLM syntax has been kept as close as possible to Java with slight variations to simplify the task of building model architecture while at the same time supporting C++ translation as well. Once translated into either Java or C++, an appropriate compiler should process the resulting code (refer to Appendix II for supported compilers). If you are already familiar with either C++ or Java you will find much of the material discussed in this chapter quite familiar, with some aspects such as modules and ports going beyond the semantics provided by either C++ or Java. If you are not familiar with either of the two languages, we recommend getting acquainted with the basic concepts found in object-oriented programming. We recommend reading one of the introductory texts such as The C++ Programming Language by Stroustrup (1997) or Core Java by Cay Horstmann and Gary Cornell (1999), among others.

This chapter is given more as a reference for the NSLM Language than a tutorial. It and the *NSLM Methods Appendix I* reviews more structures and expressions found in the language. We start by giving an overview of general aspects followed by a description of the different language components.

6.1 Overview

There are a number of general aspects in NSLM that we will overview in this section.

General Conventions

We shall be using throughout this chapter a number of general conventions used in NSLM:

- Comments are denoted by "/*" at the beginning and "*/" at the end. Single line comments are denoted with "//" at the beginning of the comment.
- All statements end with a semicolon ";".
- We consider *object types*, *object classes* or simply *classes* as equivalent terms (some programming languages distinguish between the concept of *class* and *type*). In general, *objects* represent instances of *classes*.
- We consider *module objects* as instances of *module classes*. Similarly, *model objects* represent instances of *model classes*. We treat module classes and model classes as special kinds of object classes in the programming language sense, where module objects and model objects become special kinds of objects.
- Classes—model classes, module classes and any other object classes—begin their names with an uppercase alphabetic character, e.g. *MaxSelectorModel* or *MaxSelectorStimulus*.

- Objects—model objects, module objects and objects in general—together with variables and function begin their names with a lowercase alphabetic character, e.g. *maxSelectorModel*, *maxSelectorStimulus* or *var*.
- File names storing NSLM model definitions should have a ".mod" suffix (analogous to ".C" and ".java" suffixes generated by the NSLM compiler translation).

Types

NSLM is a *typed-language*, similar to C++ and Java, supporting different types of structures. In particular, NSLM supports the following general types:

- The *primitive* or *native* data type corresponds to the basic types available in most languages, such as C and Pascal, as well as in object-oriented languages such as C++ and Java. These types always start with a lowercase letter and consist of the ubiquitous: **int**, **float**, **double**, **char** and **void** (the null type). NSLM adds two more types to this short list: **charString** and **boolean**. The **charString** type translates into "String" in Java and "char*" in C++. The **boolean** type translates into "boolean" in Java and into an enumerated type in C++ containing 1 (true) and 0 (false).
- The general *object class* data type corresponds to the basic types available only in object-oriented languages, such as C++ and Java. As opposed to the limited set of predefined *native* or *primitive* types, *object* types represent an extensible family of *classes*, either specified by the user or provided by the language in the form of libraries. The classes included in the NSLM class library are an essential component of the system and includes types, such as the scalar **NslFloat0** or the input port vector **NslDinDouble1**, used in describing neural elements, data ports or any other structure.
- The *module class* and *model class* data types corresponds to the unique family of NSLM types, distinguishing it from other object-oriented languages, such as C++ and Java. While *module classes* and *model classes* are object-oriented structures in their nature, they go beyond the basic semantics of an *object class*. *Module classes* and *model classes* incorporate semantics for input and output port based communication, something not found in "regular" object classes.

Variables, Attributes and Methods

Variables are the most basic entity in a programming language. Variables provide dual function-ality, they are used to hold either *values* (e.g. 0.5) or *references* (i.e., a virtual memory address indirectly specifying where to find the actual *values* in memory). As in most object-oriented languages, NSLM variables may not exist as independent global entities but only within an object, module or model class—being called class *attributes*. This is also the case with *functions* that may only be defined within an object, module or model class as well—being called class *methods*. Since NSLM is a typed language, every variable or attribute must first be declared according to an existing type.

- When a variable refers to a *native primitive* type, the variable will store a *value*. These types will be defined in section 6.2, *Primitives Types*.
- When a variable refers to a *module*, *model* or *class* type, the variable will store a *reference* to the particular object instead of holding a simple value. This is quite common, since objects are more complex than primitive types and thus require more sophisticated handling. (As a general comment to those users familiar with the concept of *pointers*—pointers exist in C++ but not in Java—NSLM does not include any pointer computation, only *references*.) Note that variables never refer to a class but to an object instantiated from that class. These types will be defined in sections 6.4 and 6.6, Creation of Module Types and Creation of Class Types respectively.

Attribute Reference Hierarchies

Since attributes belong to classes and attributes may provide references to other objects, we end up with attribute reference hierarchies or simply *Attribute Trees*. By providing the starting point of a tree—the root—we can access any attribute by knowing all references in its path. When NSL is running, there are two different trees present in the system:

- The *system* tree for storing NSL specific attributes, and
- The *model* tree for storing user defined attributes.

Actually, every model and module has its own attribute reference tree. If we examine one of these attribute reference trees we see that attributes of a primitive type (to be discussed below)—can only be *leaves* in the tree while attributes holding references are considered *nodes* of the tree. For example, the model attribute reference tree for the *Maximum Selector* model (as we saw in chapter 3), is shown in figure 6.1. The instance **maxSelectorModel** is the root.

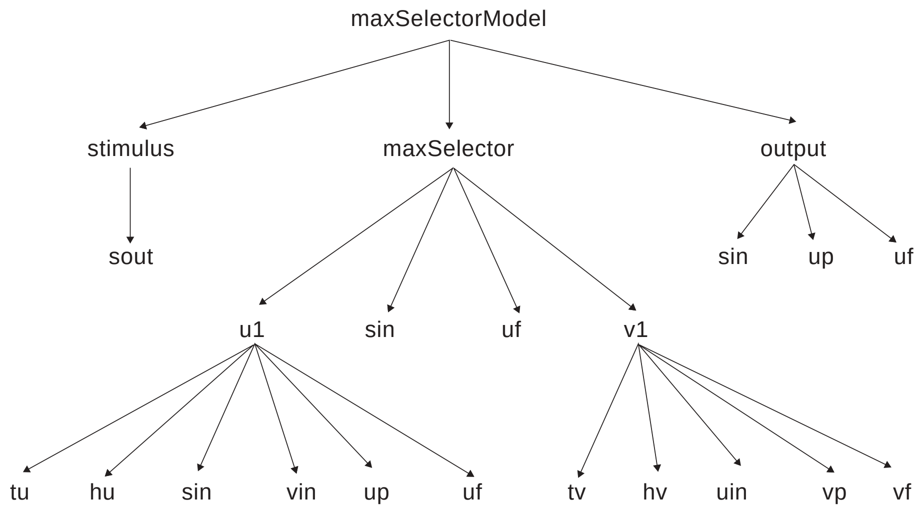

Figure 6.1
Reference tree of the *Maximum Selector* model. Note all instances of objects, or references to objects, are in lowercase, where **maxSelectorModel** is the root of the hierarchy. The model object is implicitly instantiated by NSL when simulating it (see chapter 3).

Note that every entry in the tree corresponds to a variable referencing an object and not the name of a class.

To refer to any variable we use the "dot" notation, i.e. *var1.var2*. There exist two ways of referencing a variable in the tree: *absolute* and *relative* referencing format.

- The *absolute* reference notation starts from the root of the tree specifying the complete path from there. For example, in the *Maximum Selector* model to refer to variable *vp* we must use

  maxSelectorModel.maxSelector.v1.vp

  All references for the model tree start with the name of the model.

- The *relative* reference notation starts from a particular node in the tree and continues on from there. For the previous example, if we currently reference *maxSelectorModel.maxSelector*, we would then refer to *vp* by

  v1.vp

Note that referencing a variable requires specifying its visibility as *public* (described in section 6.2) effectively breaking up the module encapsulation (where variables are defined as *private* instead). (We will also describe a similar concept called *accessibility* in section 6.3.) These concepts are analogous to how directories are made readable or not in a file system.

Class Reference Hierarchies

A different reference hierarchy also exists in all object-oriented languages. This hierarchy or tree defines references between classes as opposed to objects and it is known as the *Class Tree*. Since classes primarily exist to help define object instances, class trees are used to organize the different class definitions. The goal behind this hierarchical organization is to avoid duplicate attributes and methods by specifying common ones in the *base class* or *super class* (the class where common attributes and functions are first defined) while having other classes, known as *subclasses*, *inherit* these common attributes and methods from the *super class*. Class Inheritance Specification is quite useful in building systems and NSL takes advantage of this mechanism both internally and in letting the user build classes in general. We introduce in this section the main class hierarchy in NSL while describing how to build user-defined class trees in section 6.4 and 6.6. (Please see the NSL web site for the complete NSL Class Hierarchy details.)

Predefined Reference Variables

NSLM includes a number of pre-defined reference variables:

- **nslName** represents a **charString** type variable referencing the name of a particular object. For example, **nslName** within the *maxSelector* module object would refer to "**maxSelector**".
- **nslParent** provides a reference from any object back to where it was instantiated. For example, any reference within the **Vlayer** module instance would refer back to the **maxSelector** module instance where **Vlayer** was instantiated.
- **this** provides a reference to the current object. **this** is particularly important as a *return* reference inside functions as well as passing it as argument to another function.
- **super** provides a reference from any class back to its base class, in other words, the class from which it inherits. The **super** reference is used to retrieve attributes or methods defined in the base class, used in conjunction with inheritance. For example, a reference such as **super.***method*()would call the function *method*() defined in its base class. In general, unless specified otherwise **super** refers to the **NslModule** base module class since all modules by default inherit from it. (We will discuss this in more detail in section 6.3.).
- **null** represents the *null* or invalid reference. It is primarily used within expressions that check the validity of a reference variable.

Importing Libraries

NSLM lets the user call external libraries within a model description by using the **nslImport** statement. As in most programming languages, NSLM requires all definitions to be present during compilation. Since the user may call externally defined classes, it is necessary to include an import statement specifying what additional definitions are needed and where they should be found. For example a class defined in a file called *moreobjects.class* in Java or *moreobjects.h* in C++ would be imported using the following format,

```
nslImport moreobjects;
```

Since Java and C++ use different syntax for defining a path to an imported file, the *NSL verbatim* statement allows us to block off a section of and make it a purely Java code or C++ code. We will discuss verbatim in more detail in section 6.2.

Verbatim

NSLM includes a *verbatim* keyword telling the compiler that part of the code contains C++ or Java specific statements, as opposed to NSLM general statements, that should not be parsed by the NSLM compiler but left intact for direct C++ or Java compiler processing. For example, if the user needs more complex import statements than the ones offered in the previous section, then the *verbatim* keyword could be used. We mark *verbatim* sections in the following way:

C++-only sections of code use the notation:

```
verbatim_NSLC;
```

and Java-only sections of code use the notation:

```
verbatim_NSLJ;
```

To end the C++ or Java section of code use:

```
verbatim_off;
```

While the use of *verbatim* should be avoided, we do offer such an option when there is no appropriate NSLM construct. Note that if you use a verbatim section, the NSL preparser will not preprocess that section of code and many of the NSL constructs will need to be manually expanded to their Java or C++ correct forms. For instance all NSL library module and object instantiations actually take two extra parameters, a *charString name* and *NslModule parent*,

```
NslDouble2 someobj(“someobj”,this,4,5);
SomeModule somemod(“somemod”,this,..otherparams..);
```

We thus recommend being extremely careful when adding verbatim sections.[1]

6.2 Primitive Types

As previously mentioned, *native* or *primitive* data types store values.[2]

Defined Types

The primitive types in NSLM are shown in table 6.1. They include *numeric*, *string*, *boolean* and *void* types.

Type	Description	Initial Value
int	integer type	0
float	single precision floating point number	0
double	double precision floating point number	0
boolean	**true** or **false**	false
char	single character	null
charString	string of characters	null
void	non type	

Table 6.1
Native or *Primitive* types in NSLM with their initial default value.

It is a good practice to always set the initial value of a variable in one of the initialization methods, as will be discussed in section 6.3. It should also be noted that **void** is not actually a type but a *non-type*. It represents the lack of a type and it is used, for example, when a method returns nothing.

Declarations
Specifying the type of a variable is known as a *declaration*. In NSLM variables are declared and defined (assigning automatic space for storing values) as follows:

```
VisibilitySpec PrimitiveType varName;
```

The *VisibilitySpec* is discussed in the next section. *PrimitiveType* is the corresponding type, such as **int**; *varName* is the name of the variable storing values according to the associated type.

An example of a primitive type variable declaration with the optional initialization is as:

```
private double x = 1.0;
```

Note that we also include a *private* visibility specification in front of the declaration of *x* (described in the next section). The variable initialization is given by "=" followed by a numerical value in this case "1.0".

Visibility
Visibility specifies attributes (variables) and methods can be seen from outside a class or module. This is a very important aspect in object-oriented programming since it is the basis for *encapsulation*. Three levels of visibility are supported by NSLM, *private*, *protected* and *public*. (This is similar to visibility levels available in both C++ and Java.) The three levels are defined as follows,

- **private**—attributes and methods are local to every object instantiated from the particular class,
- **protected**—attributes and methods are local to every object instantiated from the particular class or from any of its subclasses,
- **publi**—attributes and methods are both local and external to every object instantiated from any class.

In general all attributes should be defined as *private* for encapsulation reasons. The exception to this rule is the declaration of the *ports* (to be described below) and external methods. Ports need to be available for external referencing, as do some external methods. Methods that are only used within the class should be defined as *private*. (You should be careful if omitting visibility specifications since the defaults are inconsistent between C++ and Java: C++ considers the default to be *private*, while Java considers the default to be *public* to classes defined in the same directory.) Also, the above rules only apply to attributes and methods; the visibility specification is not need if variables are declared within a method. The same visibility rules apply to object and modules types.

Arrays
NSLM supports multiple dimension arrays on all primitive types with the exception of **void**. For example a single dimension *array* of integers is defined as

```
private int alpha[10];
```

where the array *alpha* consists of 10 integers accessed each by its *index*,

```
alpha[index];
```

an integer going from 0 to the array size minus one, in this case 9.

For example, a two-dimension array or matrix is defined as

```
private float beta[10][5];
```

where the array *beta* consists of 10 by 5 float number accessed each by its *row* and *col*,

```
private beta[row][col];
```

an integer going from 0 to the array size minus one, in this case up to 9 for the *row* number and an integer going from 0, in this case up to 4, for the *col* number.[3]

Constants

The *nslConstant* keyword can be used to specify certain variables of primitive types to be constants, in other words variables that will not change over the course of the execution of the program. The syntax to do this is:

```
nslConstant VisibilitySpec type var = value;
```

For example a *public* constant would be specified as follows,

```
nslConstant public float pi = 3.14;
```

Expressions

NSLM supports primitive type expressions mainly in the form of operators similar to those common to Java and C++. Primitive expressions may be part of independent statements, passed as parameters to a method, or even as part of a return statement.

Numeric

Arithmetic operators can only be applied to numerical types—**int**, **float** or **double**—as shown in table 6.2. (Note that “unary” operators take a single argument, e.g. “*-alpha,*” while “binary” operators take two arguments, e.g. “*beta-alpha.*”)

Table 6.2
Operators that may be applied to numerical type variables, i.e., **int**, **float**, and **double**.

Operator	Usage	Description
++	++*a* or *a*++	pre-or-post increment (unary)
--	--*b* or *b*--	pre-or-post decrement (unary)
+	+*b* or *a*+*b*	positive (unary) or addition (binary)
-	-*b* or *a*-*b*	negative (unary) or subtraction (binary)
*	*a*b*	multiplication
/	*a/b*	division for doubles and floats, or modulus for integer values
%	*a%b*	remainder
=	*a=b*	assignment
*=	*a*=b*	*a=a*b*
/=	*a/=b*	*a=a/b*
+=	*a+=b*	*a=a+b*
-=	*a-=b*	*a=a-b*

For example, the assignment operator "=" assigns one number to a variable:

```
   int x;
   x = 5;
```

The first line defines the variable *x* to be of integer type, while the second line assigns a value of 5 to the variable. The two can be combined into a single statement as follows (this is known as *initialization*),

```
   int x = 5;
```

Other operators are used in a similar fashion.

Logical operators compare numerical—**int**, **float** or **double**—values to obtain a boolean type value—either **true** or **false**—as shown in the table 6.3.

Operator	Usage	Description
<	a < b	less than
>	a > b	greater than
<=	*a* <= *b*	less than or equal
>=	*a* >= *b*	greater than or equal
==	*a* == *b*	equal
!=	*a*!= *b*	not equal

Table 6.3
Logical operators that may be applied to numerical type variables, i.e., **int**, **float**, and **double**. Returns **true** or **false**.

Boolean
Boolean operators are usually seen in control statements (described in the next section): *if, while, for,* and *switch.* Operators that can be applied to boolean types are shown in table 6.4.

Operator	Usage	Description
=	*a*=*b*	Assignment among boolean values
==	*a* == *b*	Return true if the two boolean values are equal
!=	*a*!= *b*	Return true if the two boolean values are not equal
&&	*a* && *b*	Logical AND
||	*a* || *b*	Logical OR
!	!*a*	Logical NOT

Table 6.4.
Assignment and logical operators that may be applied on boolean types.

String
Operators that can be applied to charString types are shown in table 6.5.

Operator	Usage	Description
=	*a*=*b*	Copy one string to the other one
+	*a*+*b*	String concatenation
==	*a* == *b*	Return true if the two string values are equal
!=	*a*!= *b*	Return true if the two string values are not equal

Table 6.5.
Assignment, concatenation and logical operators that may be applied on string types.

Control Statements
Control statements control the flow of execution by incorporating conditions on statements. The *while*, *do* and *for* statements allows the execution flow to loop over one sec-

tion of code several times until a particular condition is met. The *if* and s*witch* statements execute a certain section of code once if a certain condition is met. NSLM includes the standard control statements shown in table 6.6. (square brackets represent optional control expressions).

Statement	Usage Example	Description

if (*condition*) { *statements* }
[**else if** (*condition*)
{ *statements* }]
[**else** { *statements* }]
	if (*a*>*b*) { *a* = 2; }
	else if (*a*>*c*) { *a* = 1; }
	else { *a* = 0; }
		if-else statement with optional intermediate *else-if* expressions and a final optional *else* expression. When *condition* is **true** the corresponding *statements* are processed.

while (*condition*) { *statements* }	**while** (*a*<*b*) { *a* ++; *c*= *a* *2; }	*while* statement. While *condition* is **true** process *statements*.

do { *statements* }
while (condition);
	do { *a* ++; *c* = *a* *2; }
	while (*a*<*b*);
		do-while statement. Process *statements* until *condition* becomes **false**.

for (*initial-expression; continuation-condition; continuation-expression*)
{ *statements* }
	for (*a* =0; *a*< *b*; *a* ++)
	{ *c* = *a* *2; }
		for statement. Execute *initial-expression*; then execute *statements* while *continuation-condition* is **true**. After each successful continuation execute *continuation-expression*.

switch (*variable*) {
case *value: statements* **break**;
[**case** *value: statements* **break**;]
[**default**: *statements*]
}
	switch (a) {
	case 0: *c* = 0; **break**;
	case 1: *c* = 2; **break**;
	default: *c* = *a*;
	}
		switch statement. Choose from the appropriate *variable* value the equivalent *value* **case** statement (as many as cases as necessary); then execute the corresponding *statements*, with an optional **default** when no matching value is found. This is equivalent to an *if-else* statement with multiple sections. At the end of each switch section a **break** statement is added.

condition?*statement-true*:*statement-false*	*c*>*d*?*a*:*b*	if *condition*, then *statement-true* else *statement-false*

Table 6.6
Control statements: *if*, *while*, *do*, *for* and *switch*.

In general, all control statements can include both a **break** and **continue** statement used to either *break* or stop processing the control statement or *continue* with the next cycle in the control statement without completing the current one. Both the break and continue statements search for the closest loop (*while, do, for*) to escape from when many nested control statements exist.

Conversions, Casting, and Promotions

As in most languages, expression return a type that can be deduced from the structure of the expression and the types of the *literals* or *operands* involved (numbers, characters, etc.). Variables may not be used in expressions where their type does not match the expected one. However, in some cases such restrictions are loosened. For example, in a method requiring an argument of type **double**, it would not be appropriate to supply a parameter of **int** type. However, languages such as C++ and Java, perform an implicit *conversion* from the deduced expression type to a type acceptable for its surrounding context, such as implicitly converting an **int** type to a **double**. In general, NSLM supports all conversions permitted by both C++ and Java on primitives, such as assignment conversion, method parameter conversion, and numeric promotion (or casting).

An assignment conversion between an integer and a float would involve an implicit *cast or conversion*:

```
private int x = 5;
private float y;
y = x;
```

where *y* stores the **float** version of *x*, in other words, 5.0.

A method invocation conversion between an integer and a float for a method defined as

```
private void func(float x) { ... }
```

would involve an *implicit cast*:

```
private int x = 5;
func(x);
```

converting *x* into a **float** when passed as an argument to the function.

Numeric promotion between an integer and a float would involve an *explicit cast or conversi*on:

```
    private int x = 5;
    private float y;
    y = (float) x;
```

where *x* gets promoted to a **float** before doing the assignment.

6.3 Object Types

NSLM *object types* or *classes* are analogous to those found in object-oriented languages, having both *attributes* (data) with corresponding *methods* (functions) to manipulate them. NSLM lets the user define new object classes as well as instantiate from a number of predefined ones, organized as *numeric*, *string*, and *boolean* classes.

Defined Types

We describe these structures followed by operators and expressions that can be applied to them.

Numeric

NSLM defines set of numeric object types varying in their dimension particularly useful for arithmetic computations. These classes vary according to the underlying numeric attribute type, **int**, **float** or **double**, and its corresponding dimension (0-4), as shown in table 6.7.

Dimension Type	0	1	2	3	4
float	NslFloat0	NslFloat1	NslFloat2	NslFloat3	NslFloat4
double	NslDouble0	NslDouble1	NslDouble2	NslDouble3	NslDouble4
int	NslInt0	NslInt1	NslInt2	NslInt3	NslInt4

Table 6.7
Numeric object types defined in NSL. The types are classified according to the corresponding dimensions, represented by the last number in the type.

The reason for providing these special types is that by encapsulating the dimension within the object, NSLM is able to *overload* a number of operators that the user would otherwise have to explicitly define. For example, additions may be specified in a single "+" statement avoiding the use of multiple "for" loops going through one primitive numeric element addition one at a time. Additionally, NSL protects the user from accessing undefined index elements (overflows), a major headache when doing direct array manipulations at the primitive level.

Boolean
NSL defines several **boolean** object types with varying dimensions as shown in table 6.8.

Dimension Type	0	1	2	3	4
boolean	NslBoolean0	NslBoolean1	NslBoolean2	NslBoolean3	NslBoolean4

Table 6.8
Boolean object types defined in NSL. The types are classified accord-ing to the corresponding dimensions, represented by the last number in the type.

As with the primitive **boolean** types the values any element of a **NslBoolean** array can hold is either **true** or **false**. **NslBoolean** object methods are discussed in more detail in the Appendix.[4]

String
NSL defines a **charString** object type useful in storing single strings of characters[5] as shown in table 6.9.

Dimension Type	0	1	2	3	4
charString	NslString0				

Table 6.9
String object types defined in NSL. The types are classified according to the corresponding dimensions, represented by the last number in the type.

The **NslString0** type, together with **charString**, are introduced in NSLM to overcome the different handling of strings in C++ and Java as well as enable the user to use strings as parameters. The methods that apply to **NslString0** types are discussed in Appendix I, *NSLM Methods*.

Ports
Ports are special object types that by linking them together enable data communication between modules—as opposed to simply storing private data within objects or modules. Ports have all the functionality defined of analogous "non-port" object types, and as such they can be used in any expression having been previously defined. Port specific expressions are described in the following sections. Ports are organized in two categories according to their semantics, *output ports* (**Dout**) and *input ports* (**Din**). We describe them according to their underlying object type.

Numeric
The numeric output and input port types are shown in table 6.10.

Dimension Type	0	1	2	3	4
float	NslDoutFloat0	NslDoutFloat1	NslDoutFloat2	NslDoutFloat3	NslDoutFloat4
	NslDinFloat0	NslDinFloat1	NslDinFloat2	NslDinFloat3	NslDinFloat4
double	NslDoutDouble0	NslDoutDouble1	NslDoutDouble2	NslDoutDouble3	NslDoutDouble4
	NslDinDouble0	NslDinDouble1	NslDinDouble2	NslDinDouble3	NslDinDouble4
int	NslDoutInt0	NslDoutInt1	NslDoutInt2	NslDoutInt3	NslDoutInt4
	NslDinInt0	NslDinInt1	NslDinInt2	NslDinInt3	NslDinInt4

Boolean
The boolean output and input port types are shown in table 6.11.

Dimension Type	0	1	2	3	4
boolean	NslDoutBoolean0	NslDoutBoolean1	NslDoutBoolean2	NslDoutBoolean3	NslDoutBoolean4
	NslDinBoolean0	NslDinBoolean1	NslDinBoolean2	NslDinBoolean3	NslDinBoolean4

String
The string output and input port types are shown in table 6.12.

Dimension Type	0	1	2	3	4
charString	NslDoutString0				
	NslDinString0				

Table 6.10
Numeric output and input port types.

Table 6.11
Boolean output and input port types.

Table 6.12
String output and input port types. (Only "0" dimension string ports are currently defined.)

Declarations and Instantiations
While classes define types, actual objects are required to exist for a program to do anything meaningful. In NSLM as in typed languages such as C++ or Java, objects are identified through variables referencing them. Specifying the type of a variable is known as a *declaration*, while actually defining the objects to which the variable refers is known as *instantiation*—creating the object for the first time. In NSLM variables are declared and have their referred object instantiated together in a single expression as follows:

```
    VisibilitySpec ObjectType varName(paramList);
```

The *VisibilitySpec* is similar to that of primitive types as discussed in the next section. *ObjectType* is the corresponding object type, such as **NslInt0**, *varName* is the name of the variable storing the reference to the new instantiated object, and *paramList* is a list of instantiation parameters that vary depending on the associated type.

Instantiation Parameters
While declarations and instantiations are similar for any NSLM or user defined types, instantiation parameters (*paramList*) vary depending on the specific type. In particular, NSLM defines certain types with a corresponding dimension suffix as shown in table 6.13.

ObjectType	*paramList*
Nsl*Type*0	
Nsl*Type*1	*size*
Nsl*Type*2	*row , col*
Nsl*Type*3	*dim, row , col*
Nsl*Type*4	*dim1 , dim2 , row , col*

Table 6.13
Instantiation parameters for the different object types in NSLM according to their dimension and corresponding suffix (0, 1, 2, 3 and 4). In the current NSL version, ***Type*** corresponds to either Int, Float, Double, or Boolean with dimensions between 0 and 4, and String for dimension 0 only. When dealing with ports, ***Type*** would include either Din or Dout together with the actual numeric type. Parameters *size*, *row*, *col*, *dim*, *dim1* and *dim2* are integer values specifying the object internal size in correspondence to the object dimension.

For example:

```
private NslInt0 x();
```

declares and instantiates an object of type **NslInt0** referenced by variable *x*. Note that NSLM automatically creates the C++ or Java code needed to allocate memory space for the new variable *x*.[6] Examples of object instantiations for different dimensions and types are,

```
private NslDouble1 y(10);
private NslFloat2 z(10,5);
```

where 10 is the *size* of object *y* and 10 by 5 is the size (*row*,*col*) of object *z*.

In the case of ports, the syntax to declare and instantiate a port type is similar to that used for normal NSL numeric types, namely:

```
public NslDinInt0 xp();
```

declares and instantiates an object of type **NslDinInt0** referenced by variable *xp*. Examples of object instantiations for different dimensions and types are,

```
public NslDinDouble1 yp(10);
public NslDoutFloat2 zp(10,5);
```

where 10 is the *size* of object *yp* and 10 by 5 is the size (*row*,*col*) of object *zp*. Also note that we always include the visibility declaration of "public" since other modules need to be able to connect to these ports. Ports should only be defined as attributes and not within methods.

There is an alternative option for defining objects without fully assigning its internal size during instantiation. This is particularly useful when providing array sizes in a dynamic way. The format is as follows:

```
private NslDouble1 r();
private NslFloat2 s();
```

where no specific values are given for the *r* or *s* corresponding dimensions. The dimensions are set at a later time using the *var***.nslMemAlloc(***sizeList***)** method (see Appendix I for further details) where *sizeList* represents the corresponding sizes from the original *paramList* in one of the constructor or initialization methods discussed in section 6.3. Defining objects this way provides great flexibility since models may have their internal sizes dynamically assigned during model execution avoiding recompilation such as in the extensions mentioned for the *Backpropagation* model described at the end of chapter 3.

Arrays

Array usage with object types varies from that of primitive types. In the current NSLM version the user may define arrays of primitive types but they may not define object type arrays. This is because of the differences of array handling in Java and C++. Yet, the NSL C++ version does support C++ arrays of NSL types as described in Appendix III (NSLC extensions). On the other hand, since NSL predefined object types already include up to four dimensions, NSL includes array-style data accessing in those types.

In general, accessing data from an object is done through method invocations or in the case of NSL dimensional objects, it varies depending on the particular dimension objects may have. To access an element within an object, NSLM provides array indexing using the conventional bracket pair "[ ]" according to the following considerations (note that all array indices start at zero).

- **Zero:** A zero-dimensional type stores a single primitive type value or scalar. For example **NslFloat0** type stores a **float** type.
- **One:** A one-dimensional type stores a one-dimensional primitive type array (considered a row vector). For example **NslFloat1** type stores a **float** type array. In a one-dimensional object *m*, *m*[*j*] returns the *j*+1-th element in *m*, where *j* must be a positive integer value (or an expression returning a positive integer value).
- **Two:** A two-dimensional type stores a two-dimensional primitive type array (considered a matrix). For example **NslFloat2** type stores a **float** type two-dimensional array. The first dimension of the array represents the rows while the second dimension represents the column. In a two-dimensional object *m*, *m*[*i*] returns the *i*+1-th row of the array, which is a one-dimensional array. *m*[*i*][*j*] returns the element at the *i*+1-th row and *j*+1-th column of the array.
- **Three:** A three-dimensional type stores a three-dimensional primitive type array (considered a vector of matrices). For example **NslFloat3** type stores a **float** type vector containing two-dimensional arrays. The left-most dimension identifies the vector while the other two represent the rows and columns of the matrix, respectively. In a three-dimensional object *m*, *m*[*h*] returns the *h*+1-th two-dimensional array. *m*[*h*][*i*][*j*] returns the element at the *h*+1-th array, *i*+1-th row and *j*+1-th column of the two-dimensional array.
- **Four:** A four-dimensional type stores a four-dimensional primitive type array (considered a vector of three-dimensional matrices). For example **NslFloat4** type stores a **float** type array of a **float** type array that stores a two-dimensional arrays. In a four-dimensional object *m*, *m*[*g*] returns the *g*+1-th three-dimensional array, *m*[*g*][*h*] returns the *g*+1-th and *h*+1-th two-dimensional array. *m*[*g*][*h*][*i*] returns the *g*+1-th, *h*+1-th, *i*+1-th vector. *m*[*g*][*h*][*i*][*j*] returns the *g*+1-th, *h*+1-th, *i*+1-th , *j*+1-th element of the array.

(A number of methods manipulation objects with different dimensions are described in Appendix I.) table 6.14 summarizes array indexing and partial indexing.

ObjectType	**Indexing**	***ResultType***
Nsl*Type*1	*var[index]*	*type*
NslType2	*var[row]*	**Nsl*Type*1**
	var[row] [col]	*type*
NslType3	*var[index1]*	**Nsl*Type*2**
	var[index1] [row]	**Nsl*Type*1**
	var[index1] [row] [col]	*type*
NslType4	*var[index1]*	**Nsl*Type*3**
	var[index1] [index2]	**Nsl*Type*2**
	var[index1] [index2] [row]	**Nsl*Type*1**
	var[index1] [index2] [row] [col]	*type*

Table 6.14
Indexing for dimensional object types. ***Type*** corresponds to either **Int**, **Float**, **Double**, **Boolean** or String for their correspondingly defined dimensions (similarly with port types). Again, *index*, *row*, *col*, *index1* and *index2* are integer values specifying the corresponding index number. *type* would correspond to a primitve type either **int**, **float**, **double**, **boolean** or **charString** respectively.

Constants
Similar to primitive types, the *nslConstant* keyword can be used in conjunction with object types in order to for them to be constants, in other words variables that will not change over the course of the execution of the program. The syntax to do this is:

```
nslConstant visibilitySpec objectType varName(paramList) =
value;
```

For example a public constant would be specified as follows,

```
nslConstant public NslFloat0 pi = 3.14;
```

Expressions
NSLM supports a number of expressions on the different defined types. We will described them according to *numeric*, *boolean*, *string* and *port* types.

Numeric
NSLM supports most numeric operators to those defined for primitive numeric types (this applies to both numeric and numeric port types). The supported arithmetic expressions are shown in the table 6.15.

Operator	**Usage**	**Description**
=	*a*=*b*	Assignment
+	+*b* or *a*+*b*	Unary Positive or Two Parameter Addition
-	-*b* or *a*-*b*	Unary Negative or Two Parameter Subtraction
/	*a*/*b*	Pointwise Division
^	*a*^*b*	Pointwise Multiplication
*	*a***b*	Scalar Multiplication or Vector/Matrix Product
@	*a*@ *b*	Vector/Matrix Convolution (see Appendix II)

Table 6.15
Arithmetic operators for object numeric types. Typically, operands required on each side of the operator (except in the case of the unary operators) must be of a similar type and dimension although in some cases parameter may be of different dimension. More details on valid operands are described in Appendix II.

For instance, a NSL **NslFloat0** object would have its value assigned as follows:

```
private NslFloat0 x();
x = 5.0;
```

If the assignment takes place on the same line as the declaration and instantiation then it is known as *initialization*:

```
   private NslFloat0 x() = 5.0;
```

A NSL **NslFloat2** object would have one of its elements assigned a value as follows:

```
   private NslFloat2 y(2,3);
   y[0][1] = 5.0;
```

or all its elements as follows,

```
   y = 5.0;
```

where 5.0 is assigned to every element in *y*.

Note that assignment copies values from one object to another one. This is not a copy of references (as opposed to Java handling of object to object assignment). For example

```
   private NslFloat2 z(2,3);
   z = y;
```

assigns every element value in *y* to every element value in *z*, where *y* and *z* must be equally sized.

Also note that numeric objects on the left-hand side of an assignment statement may be assigned with primitive types returned on the right-hand side. For example, the following code works without having to add an explicit cast. (Explicit cast is covered in this section on *Conversions, Casting, and Promotions*.)

```
   private NslFloat1 phi(5);
   private NslFloat0 force();
   private float mu;
   phi=22;
   mu=phi[0];
   force=phi[0];
```

The previous to last equation copies the content of *phi*[0] into *mu*, while the last statement copies *phi*[0] to *force*.

NSLM also provides logical operators for numeric port types. The logical operators are shown in table 6.16.

Operator	Usage	Description
<	a < b	less than
>	a > b	greater than
<=	*a* <= *b*	less than or equal
>=	*a* >= *b*	greater than or equal
==	*a* == *b*	equal
!=	a!= b	not equal

Table 6.16
Logical operators for object numeric types.

All logical operators are applied as pointwise to arrays and return an array of similar size. If the arrays are of different dimension or size, an error will occur.

Boolean

Boolean types are mainly used as resulting values from statement conditions. For example all expressions in table 6.15 return a boolean value. Since boolean values can only be true or false, the only expression that can be applied to this values are the ones shown in table 6.17.

Operator	Usage	Description
=	*a=b*	Assignment among boolean values
==	*a == b*	Return true if the two boolean values are equal
!=	a!= b	Return true if the two boolean values are not equal
&&	*a && b*	Logical AND
||	*a* || *b*	Logical OR
!	! *a*	Logical NOT

Table 6.17
Logical operators that may be applied on boolean expressions.

String

String type expressions are shown in table 6.18.

Operator	Usage	Description
=	*a=b*	Copy one string to the other one
+	*a + b*	String concatenation
==	*a == b*	Return true if the two string values are equal
!=	*a*!= *b*	Return true if the two string values are not equal

Table 6.18
Logical operators that may be applied on boolean expressions.

For example, to declare as well as initialize a variable of type NslString0 we type:

```
private NslString0 protocol()= “Protocol”;
```

Ports

In general all “non-port” expressions apply to port types, i.e., numeric type expressions apply to numeric port types, similarly with booleans and strings. There a number of additional expressions, in the form of methods particular to port types defined for specifying *connections* and *relabels* between them. When connecting or relabeling ports the type and dimension of the ports must match or a compilation error will occur, the only exception is connecting among different numeric port types. In order to illustrate these expressions in more detail we present in figure 6.2 a comprehensive diagram of the *Maximum Selector* model previously described in chapter 3.

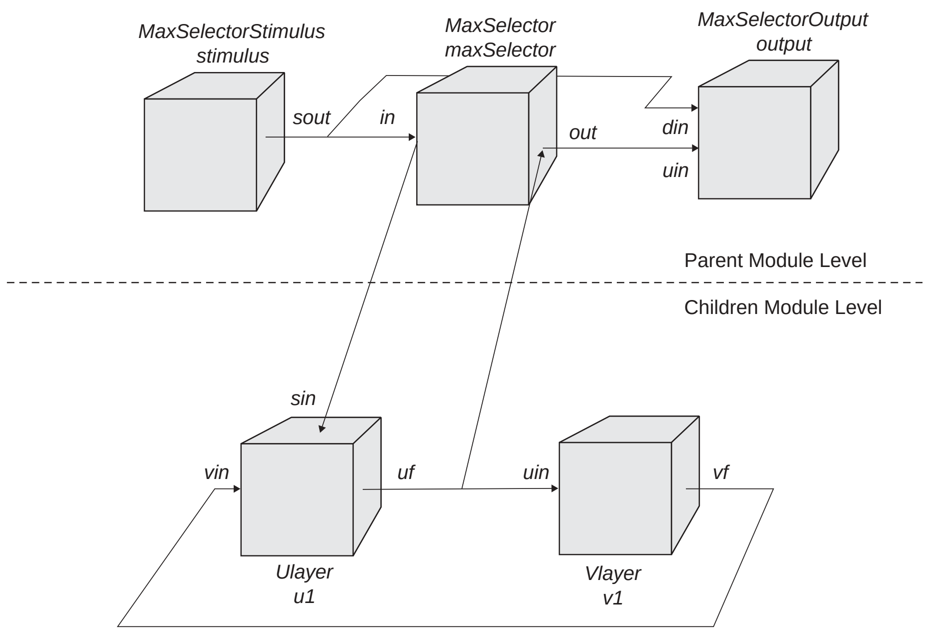

Figure 6.2
Complete diagram representation for the *Maximum Selector* presented in chapter 3. The model contains a top-level (*Parent Module Level*) consisting of the the **MaxSelecto**r module and both the **MaxSelectorStimulus** and **MaxSelectorOutput** modules. The **MaxSelector** module is decomposed at the bottom-level (*Children Module Level*) into the **Ulayer** and **Vlayer** modules. Connections are made between modules at the top-level or bottom-level, respectively. Relabels are made across modules in the top-level and bottom-level.

Connections

NSL provides the user with a special function nslConnect to make connections between ports. Connections are always specified from output ports to input ports as follows,

```
   nslConnect (m1.dout,m2.din);
```

where output port *dout* in module *m1* is connected to input port *din* in module *m2*. This statement, specified within the **makeConn** method, shows how to specify port connections between modules and should be specified at the parent module level, that is inside the module that actually instantiated both *m1* and *m2*. For example, at the *parent module level* the following three connections are made in figure 6.2,

```
   nslConnect(stimulus.sout,maxselector.in);
   nslConnect(stimulus.sout,output.sin);
   nslConnect(maxselector.out, output.uin);
```

At the *children module level* the following two connections are made in figure 6.2,

```
   nslConnect(v1.vf,u1.vin);
   nslConnect(u1.uf,v1.uin);
```

As an additional consideration, in order to refer to a port it must have been defined with visibility public, otherwise the above connection statement would cause a compilation error.

Relabels

Besides connections NSL provides a special connectivity function **nslRelabel** to forward data between ports belonging to modules at different levels in the module tree hierarchy. Recall that connections are done between an output port and an input port belonging to different modules at the same decomposition level. On the other hand, relabeling is specified between a *parent module* input port and a *child module* input port or between a *child module* output port and a *parent module* output port, respectively. Relabeling plays an important role when building module *compositions* or *assemblages*. For example, the following statements would relabel an input port *din* at the parent module level to *din* in *m1* and an output port *dout* in *m1* to *din* at the parent module level,

```
nslRelabel(din,m1.din);
nslRelabel(m1.dout,dout);
```

For example, the following two relabels are made in figure 6.2,

```
nslRelabel(in,u1.sin);
nslRelabel(u1.uf,out);
```

Note that we do not need a module reference in the first argument since the reference is the actual module where the relabel is taking place—the "**this**" module.

Control Statements

Previously defined control statements (see "Primitive Types" section) support the use of object types as long as the corresponding expressions allow it. For example, boolean conditions permit the use of object type expressions returning boolean types. More generally, statement accept any object type defined expressions.

Conversions, Casting, and Promotions

Analogous to primitive types, object type variables may not be used in expressions where their type does not match the expected one. However, in some cases such restrictions are also loosened. For example, in a method requiring an argument of type **NslDouble0** a **NslInt0** type would be accepted as well. This applies between all numeric types as long as their dimension corresponds. In particular port types can be used whenever "non-port" types correspond. (The opposite is not true since "non-port" types cannot be connected among themselves.) No conversions, castings or promotions may be applied between primitive and object types.

6.4 Creation of New Object Types

NSLM allows for the creation of new object-oriented style classes (creation of new module and model types is described in section 6.5 and 6.6 respectively). As will be seen later, the major differences between modules and classes is their lack of ports and any control from the NSL scheduler. Defining a new class involves defining attributes and methods common to all its objects. New objects may be instantiated only if there exists a previously defined corresponding class as in most object-oriented languages. The class definition format is somewhat similar to that in C++ or Java. (The user can also define native C++ or Java classes with the use of the verbatim modifiers.)

Template

To define a new class we use the special **nslClass** keyword in the class definition header, as shown in code segment 6.1.

```
nslClass class-name ( class-instantiation-spec ) class-
inheritance-spec
{
   class-attribute-spec
   class-method-spec
}
```

Code Segment 6.1
nslClass definition template.

The code section outside the curly brackets corresponds to the class *header*, consisting of the following:

- *class-name* represents the name identifying the class.
- *class-instantiation-spec* defines instantiation arguments (type-name pairs separated by commas) that must be passed when instantiating a new object. (This corresponds to the header of the object constructor in either C++ or Java.)
- *class-template-inheritance-spec* defines the inheritance specification for the class.
- The code section that appears inside the curly brackets defines the actual structure and functionality of the class:
- *class-attribute-spec* defines the structure of the class in terms of its attributes, primitive and object type variables.
- *class-method-spec* defines the behavior of the class in terms of local function or method definitions.

Header
The basic class header includes the **nslClass** keyword, the *class-name*, and the *class-instantiation-spec*. For example, code segment 6.2 describes the main header section for a **MemoryCalc** class.

```
nslClass MemoryCalc (int size) class-inheritance-spec
{
   class-attribute-spec
   class-method-spec
}
```

Code Segment 6.2
Class header specification.

The *class-instantiation-spec* defines a single instantiation parameter *size* passed to class **MemoryCalc**.

Inheritance
Every class definition in NSLM requires a *class-inheritance-spec*. Inheritance is an important feature present in all truly object-oriented languages permitting the definition of new classes as extensions to already existing ones. The inheritance scheme provides the new class with all the attributes and methods (except for the *private* ones) of the *superclass* or *baseclass*, where the new class is known as the *subclass*. Inheritance is also the basis for code reuse in an application. As an aside for those users familiar with the concept, NSL supports only single inheritance—as opposed to multiple inheritance. In order to show how the *class-inheritance-spec* is used, we will create another class called *MovementCalc* that inherits from *MemoryCalc* as shown in code segment 6.3.

```
nslClass MovementCalc (int size) extends MemoryCalc (size)
{
   class-attribute-spec
   class-method-spec
}
```

Code Segment 6.3
MovementCalc class header with inheritance from MemoryCalc.

We define **MovementCalc** as a subclass of **MemoryCalc**. Recall that **MemoryCalc** requires an instantiation argument. Thus, we must pass *size* to **MemoryCalc** as shown in the inheritance specification to avoid an error. Note the difference between the instantiation argument specification containing the type of the argument—int for *size*—and the parameter passed to the base class containing only the parameter name and not its type.

Also note that **MovementCalc** can use all of the *public* and *protected* attributes and methods from **MemoryCalc** class.

NSLM provides a default empty inheritance specification on the previous **MemoryCalc** class definition as shown in code segment 6.4. In this case, the NSLM compiler generates the default inheritance specification "**extends NslClass**(**charString** nslName, **NslModule nslParent**)" when none is provided. **NslClass** is directly or indirectly the highest superclass for all class objects.

```
nslClass MemoryCalc (int size)
{
   class-attribute-spec
   class-method-spec
}
```

Code Segment 6.4
Class header with empty inheritance specification.

Attributes
Attributes define the structure of the class. Attributes may be either primitive or object types. For example, in code segment 6.5 we add a private object to the **MovementCalc** class.

```
nslClass MovementCalc (int size) extends MemoryCalc (size)
{
   private NslInt1 vector1(size);
   class-method-spec
}
```

Code Segment 6.5
Example of a class variable attribute.

The attribute section consists of a **private NslInt1** type object referenced by variable named *vector1*, with instantiation argument *size* corresponding to the vector size. Variable *size* is part of the instantiation arguments. Note that we should not name attributes the same as instantiation arguments since that will cause a compilation error. Furthermore, since we also have a base class containing its own attributes and methods, we must avoid conflicts with attributes with similar names in base classes.

Methods
Methods or functions define the behavior of the class. Methods correspond to functions in structured languages such as C and directly correspond to those defined in object-oriented languages. Methods must always be defined within a class corresponding to the *class-method-spec* section. The body of a method—its implementation—supports expres-sions and statements similar to those used in C++ or Java, involving both primitive and object types. Methods can take any number of parameters and may or may not have a return type. Both arguments and the return type may be either objects or primitive types. As an example, we add a *print* method to the **MemoryCalc** example as shown in code segment 6.6.

```
nslClass MovementCalc (int size) extends MemoryCalc (size)
{
   private NslInt1 vector1(size);
   public int print() {
      nslPrint("vector1:",vector1);
   }
}
```

Code Segment 6.6
Example of a class method.

This simple *print* method prints the values stored in *vector1*. To actually call the method we use the “dot” notation. For example, if we want to call the *print* method from within this or other class we would do the following:

```
MemoryCalc m();
m.print();
```

Similar to the **nslPrint** method NSLM also provides a wide number of methods for arithmetic calculations, file manipulation and other functionality as described in Appendix II.

Static Modifier

NSLM offers an additional **static** modifier affecting both attributes and methods. The modifier makes the previously defined *object attributes* and *object methods* become what is known as *class attributes* and *class methods* respectively. The difference between the two lies in that object attributes and methods are designed to be accessed by an object reference where every object from a particular class refer to different data (with similar or different values) for the same attribute. On the other hand, class attributes and methods are designed to be accessed by a class reference where all objects from a particular class refer to a common data with a unique value for the same attribute. In other words a class attribute is an attribute whose value is always the same to all objects instantiated from that class, i.e. each object does not have its own private copy of the attribute but a shared one with all other objects. For example, in code segment 6.7, we show how to define a class attribute and a class method for **MemoryCalc**.

```
nslClass MemoryCalc (int size) {
    private static int version;
    public static int print() {
        nslPrint(“MemoryCalc“);
    }
}
```

Code Segment 6.7
Example of the use of the **static** keyword

The method *print* is used to print the name of the class as opposed to the name of an object instance. The method is called using the class name as its reference

```
MemoryCalc.print();
```

This is quite useful in defining libraries that perform transformations dependent exclusively on data passed to it, such as with numerical functions. In this case no objects need to be instantiated from that class in order to execute the function. (In general objects are instantiated to store data for future use. In the case of simple transformation no “memory” is required and functions perform direct transformations based exclusively on arguments currently passed to it.)

6.5 Creation of New Module Types

Modules are the basis for processing and simulation in NSL. Modules are the most important NSL structure, distinguishing NSL from being “just another” object-oriented language. Modules are concurrent or *active* entities with the potential to be distributed[7] based on communication *ports* for sending and receiving data between modules. This is in addition to traditional object-oriented message passing between objects in the form of method invocations. Thus, modules are distinguished from object classes in that module methods are executed by the NSLM scheduler, whereas object methods have to be explicitly called by the user.

Template

The process of defining new modules is similar to that of object classes. A module *template* is defined having module attributes and methods similar to those defined for an object class in addition to specific module port attributes and simulation methods. In terms of syntax, modules use the special **nslModule** keyword instead of **nslClass**. The module definition template is shown in code segment 6.8.

```
nslModule module-name ( module-instantiation-spec ) module-
inheritance-spec
{
   module-attribute-spec
   module-method-spec
}
```

Code Segment 6.8
nslModule definition template.

The template section that appears outside the curly brackets corresponds to the module header, consisting of the following

- *module-name* is the name identifying the module.
- *module-instantiation-spec* defines instantiation arguments that must be passed when creating a new module instance.
- *module-inheritance-spec* defines class inheritance aspects for the module.

The template section that appears inside the curly brackets defines the actual structure and functionality of the module:

- *module-attribute-spec* defines the structure of the module in terms of primitive, object and module type variables, including ports necessary for external communication.
- *module-method-spec* defines the behavior of the module in terms of local functions or methods definitions, including simulation methods.

Header

The module-instantiation-spec within the header defines arguments that must be passed when instantiating a new module similar to class templates in NSLM. The specification is made of a list of type-name pairs separated by commas that may also be empty. For example, code segment 6.9 shows two instantiation parameters in **BasicModule**.[8]

```
nslModule BasicModule (int size, NslString0 c) module-
inheritance-spec
{
   module-attribute-spec
   module-method-spec
}
```

Code Segment 6.9
BasicModule header specification.

Inheritance

The *module-inheritance-spec* allows the module to inherit attributes and methods from a *base module class* or *super module*. Module inheritance is similar to that in regular classes except that all modules must inherit directly or indirectly from **NslModule** in order for modules to be correctly managed. If we want to define a new module type that inherits from one already created, then we must pass the required parameters to the super module class as shown in code segment 6.10. Since **BasicModule** requires both *size* and *c*, we pass them as parameters from **ExtendedModule**.

```
nslModule ExtendedModule (int size, NslString0 c, char ptype)
extends BasicModule(size,c)
{
   module-attribute-spec
   module-method-spec
}
```

Code Segment 6.10
ExtendedModule module header with inheritance from BasicModule.

If inheritance is not specified, as in code segment 6.11, the NSLM compiler automatically appends the code "**extends NslModule(charString** nslName, **NslModule** nslParent)" to the header.

```
nslModule BasicModule (int size, NslString0 c)
{
   module-attribute-spec
   module-method-spec
}
```

Code Segment 6.11
BasicModule header with empty inheritance specification.

Also note that all modules inherit directly or indirectly from a class called **NslModule** in order to take advantage of attributes and methods such as getting the variable's name, getting the variable's parent, setting the script access to the variable, and printing the variable. Similar to class, NSL supports only single inheritance for modules.

Attributes
Attributes define the structure of the module. As in object classes, attributes may be either primitive, object or module types. For example, we add single input and output port to the **ExtendedModule** module structure, as shown in code segment 6.12.

```
nslModule ExtendedModule (int size, NslString0 c, char ptype)
extends BasicModule(size,c)
{
   public NslDinFloat1 din(size);
   public NslDoutFloat1 dout(size);
   module-method-spec
}
```

Code Segment 6.12
Example of module port attributes.

The attribute section consists of,

- **public NslDinFloat1** input port named *din*, with instantiation parameter *size* since it corresponds to a numeric vector.
- **public NslDoutFloat1** output port named *dout*, with instantiation parameter *size* since it corresponds to a numeric vector.

Methods
Modules are different from objects in their incorporation of simulation methods in addition to object type style methods. In this section we describe simulation methods followed by differential equation methods, of particular importance to modules.

Simulation
Simulation methods are executed during system runtime according to control parameters specified by the user from the script or window interpreter. Simulation methods, in addition to class methods, are inserted into *module-method-spec* section. In code segment 6.13, we define one protected method and three public methods. Two of these methods override two NSLM's predefined simulation methods, **initRun** and **simRun**, respectively.

(The **initRun** and **simRun** are discussed below; however, for a complete list of NslModule methods please see Appendix I, NSLM Methods.)

Code Segment 6.13
Example of modules methods.

```
nslModule ExtendedModule (int size, NslString0 c, char ptype)
extends BasicModule(size,c)
{
   public NslDinFloat1 din(size);
   public NslDoutFloat1 dout(size);
   public double getVelocity(int deltax, int deltay) {
      //more code
   }
   public void initRun() {
      dout=0;
   }
   public void simRun(){
      //more code
   }
   protected NslDouble2 eyeMoveSpecial(int deltax, int deltay)
   {
      //more code
   }
}
```

All simulation methods are defined in the class **NslModule** and are *overridden* by the user through similarly named methods in the new module. Note that many of these methods were given in chapter 3 together with examples. The following tables describe the available methods for *overriding* (all methods return a **void** type and have no arguments passed to them). table 6.19 shows the *connection* method.

Table 6.19
Connection method.

Connection Method	Description
makeConn	All connections and relabels between modules should be specified within this method.

Table 6.20 shows the *system* methods called once throughout the execution of the complete system.

System Methods	Description
initSys	This method should contain any initializations required for the complete system (one per module). This usually involves system variable initialziations.
endSys	It is the last method called before the end of the complete system simulation, for example to execute any summary type calculations.

Table 6.20.
System methods.

Table 6.21 shows the *module* methods called once during a complete module simulation.

Module Methods	**Description**
initModule	Initializes a module during every simulation, both training and run phases. For example, the number of simulation epochs or cycles per epoch may be set here.
endModule	Ends the complete simulation, both training and run phases. Performs any simulation post-processing.

Table 6.21 Module methods.

Table 6.22 shows the *train* methods called in relation to training aspects of the simulation.

Train Methods	**Description**
initTrainEpochs	Initializes variables that are needed for all train epochs.
endTrainEpochs	Summarizes the results from all train epochs.
initTrain	Initializes the training phase for all train cycles and is executed once per train epoch. Training variables are reset in this method.
simTrain	Contains training dynamics. Simulates the training phase for as many steps as specified or until reaching **trainEndTime** divided by **trainDelta**.
endTrain	Executes at the end of the training phase for a single step when the time step corresponds to **trainEndTime**. Usually used for compiling statistics and printing results after each train epoch.

Table 6.22 Train methods.

Table 6.23 shows the *run* methods called in relation to running aspects of the simulation.

Run Methods	**Description**
initRunEpochs	Initializes the variables that are needed for all run epochs.
endRunEpochs	Summarizes the results from all run epochs.
initRun	Initializes the run phase for al run cycles and is executed once per run epoch. Variables for the run are reset in this method.
simRun	Contains running dynamics. Simulates the run cycle for as many steps as specified or until reaching **runEndTime** divided by **runDelta**.
endRun	Executes at the end of the run phase for a single step when the time step corresponds to **runEndTime**. Usually used for compiling statistics from each run and printing some kind of results. It may include modifications on the simulation parameters.

Table 6.23 Run methods.

Note that it is not mandatory to redefine or override any of these methods. When not overridden, the default method within the direct superclass (or **NslModule** by default) will be called. Also, the simulation time in a **simTrain** phase or a **simRun** phase starts with time equal zero and changes by time equal **trainDelta** or **runDelta** after each cycle or step. The number of cycles and the number of epochs both start at one when time equals zero.

Since NSLM controls the scheduling of module methods, these should not be directly called from user-defined expressions or control statements.

Differential Equations

Differential equations are quite important in modeling neural networks. Simulation of neural networks as introduced in chapter 1 is based in NSL on the leaky integrator neural model specified by a first-order differential equation of the form

$$\tau \frac{dmp_t}{dt} = f\left(t, mp_t\right) \tag{6.1}$$

This first-order differential equation requires the use of numerical approximations to solve it. NSLM provides a general method for first-order differential equations defined as follows:

```
nslDiff(mp,τ,f(t,mp));
```

or

```
mp = nslDiff(mp,τ,f(t,mp));
```

where *f*(*t*,*mp*) represents any mathematical expression, for example

```
f(t,mp) = -mp+s;
```

corresponds to the leaky integrator model where *s* represents to the neuron input. *τ* is a time constant having default value 1.0, and *dt* is the time delta. Since *dt* is not specified, its value is given from the script command interpreter.

While different numerical methods may be used to solve the equation, NSLM defines it in such a way that the actual neural network architecture and connections do *not* change when changing the aproximation method used. Different numerical methods may be more or less appropriate according to the desired numerical precision and the processing power of the computing machine. NSLM includes two approximation methods, *Euler* and 2nd order *Runge-Kutta*, specified by

```
setApproxMethod(method);
```

where *method* can be either the string **Euler** or **RungeKutta2**.

- The difference equation specified by the *Euler* approximation method is:

$$\frac{\tau\left(mp_{t+\Delta t} - mp_t\right)}{\Delta t} = f\left(t, mp_t\right) \tag{6.2}$$

and is modified to

$$mp_{t+\Delta t} = mp_t + \left(\frac{dt}{\tau}\right) f\left(t, mp_t\right) = mp_t + \left(\frac{dt}{\tau}\right)\left(-mp_t + s_t\right) \tag{6.3}$$

or expanding the equation for the leaky integrator,

$$mp_{t+\Delta t} = \left(1 - \frac{\Delta t}{\tau}\right) mp_t + \frac{\Delta t}{\tau} s_t \tag{6.4}$$

- The difference equation specified by the *Runge-Kutta2* approximation method is expanded into*:*

$$h = \frac{\Delta t}{\tau} \tag{6.5}$$

$k_1 = hf(t, mp_{t,})$(6.6)

$$k_2 = hf\left(t + \frac{1}{2}h, mp_t + \frac{1}{2}k_1\right) \quad (6.7)$$

$$mp_{t+\Delta t} = mp_t + k_2 \quad (6.8)$$

or expanding the equation again,

$$mp_{t+\Delta t} = \left(1 - \frac{(\Delta t)^2}{4\tau^2}\right) mp_t + \frac{\Delta t}{2\tau} s_t \left(1 + \frac{\Delta t}{2\tau}\right) \tag{6.6}$$

Scheduling

NSL provides a multi-clock scheduler for each module during simulation. Every train cycle executes the **simTrain** method **trainDelta** times for as many cycles as specified by **numTrainEpochs**. Similarly, every run cycle executes the **simRun** method **runDelta** times for as many cycles as specified by **numRunEpochs**. The detailed order of execution including initializations is as follows:

1. For all modules execute **initSys**.
2. For all modules execute **makeConn**.
3. For all modules execute **initModule**.
4. Execute simulation cycles for as many epochs as specified, both train and run.
5. For all modules execute **endModule**.
6. For all modules execute **endSys**.

The controls these cycles by using either the menu commands from the NSL Executive window or the NSLS script commands in the script window (see chapter 5). However, **initSys**, **makeConn**, and **initModule** will all be called before the NSL Executive window and the script window appear. Figure 6.3 shows in more detail a flowchart corresponding to step 4: the train and run phases. (Usually the train phase is executed before the run phase.)

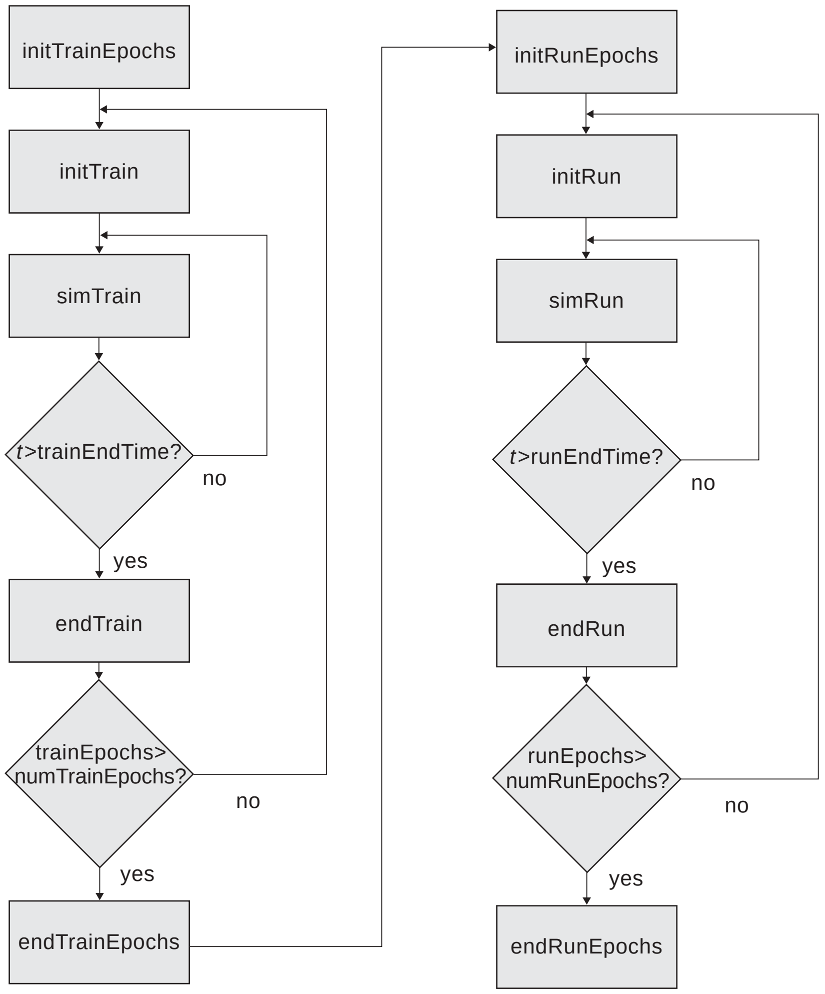

Figure 6.3
Flowchart showing train and run processing phases. Note that the outer loop is referred to as the "epoch" and the inner loop is referred to as the "cycle".

The scheduling described above is applied sequentially to all modules. In other words no module will execute a **simTrain** method unless all modules have previously executed an **initTrain** method. If a module does not define a particular method (say **initRun**), then that module will simply be skipped when its turn comes up for that phase of execution. The order in which modules are processed for a single method pass is **preorder** starting with the main model in the attribute reference tree hierarchy, as exemplified in figure 6.4. The simulation sequence is generated by going over all modules according to their initial instantiation specification, i.e. as soon a module is instantiated it is immediately put into the scheduling list. In figure 6.4, the scheduler would start from the top module **ModuleA** followed by its first child **ModuleB**. Since **ModuleB** has children, then the scheduling list continues with **ModuleC** and son on. The complete execution order in this example is alphabetical, i.e. **ModuleC** followed by **ModuleD** then followed by **ModuleE** and so on.[9] Note that the tree is implicitly built by NSL following the order of module instantiations by the user.

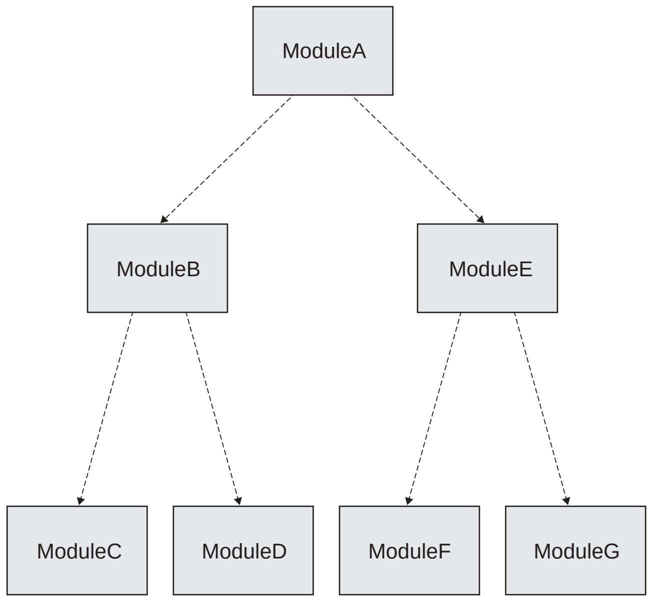

Figure 6.4
Dashed arrows in the frame specify instantiation order, starting with the top module in preorder and depth first fashion, in other words, following names "alphabet-wise."

The actual code to generate the tree shown in figure 6.14 is as follows (note the exact declaration order with the modules. The order of module definition templates is unimportant).

```
nslModule ModuleA ()
{
   ModuleB b();
   ModuleE e();
}
nslModule ModuleB ()
{
   ModuleC c();
   ModuleD d();
}
nslModule ModuleE ()
{
   ModuleF f();
   ModuleG g();
}
```

Code Segment 6.14
Module code definitions generating the hierarchy tree shown in figure 6.4. Note however, that each module must be declared in its own file. Thus the above example would take three files.

Buffering

To simulate concurrency in module execution NSL offers buffered ports as well as non-buffered (default) ports. In the default mode, ports are non-buffered and processing becomes sequential processing where new values from output ports are immediately sent to all ports connected to it. In this form, ports operate as a numeric object keeping values for internal and external use always the same. Concurrency is only simulated with buffered ports making processing order unimportant since all communication becomes buffered and output values are not immediately sent out from a module's output ports. With buffered ports, the buffer keeps temporary internal port values. After each simulation cycle the system copies the all buffered values into a second buffer used for communications with other modules. Since the default mode is set to non-buffered ports, the user may change a particular port to the buffering mode with the following command:

```
port.nslSetBuffering(true);
```

If instead of true the argument is **false** then the port becomes non-buffered. Additionally, NSL offers the following command to set all ports in a module to the buffered state:

```
module.nslSetBuffering(true);
```

To make all ports in all modules buffered the user may use the following command (again a **false** would reset this mode):

```
system.nslSetBuffering(true);
```

When dealing with buffered ports, the system internally executes a method named **nslUpdateBuffers** after each simulation cycle to update port buffers.

To get the current buffer setting we can call one of the following methods returning a *fg* boolean value:

```
fg=system.nslGetbuffering();
fg=module.nslGetBuffering();
fg=port.nslGetBuffering();
```

6.6 Creation of New Model Types

A *model* defines a complete program or application. Instead of having a "main" function as required in most programming languages, NSL applications require the existence of a *model* where execution begins. This *model* is unique to every NSL application. Its definition is somewhat similar to module definitions except in the use of the **nslModel** keyword (instead of nslModule) as shown in code segment 6.16. Note that NSLM does not allow any instantiation parameters nor a inheritiance specification for models.[10]

```
nslModel model-name ()
{
   model-attribute-spec
   model-method-spec
}
```

Code Segment 6.15
nslModel Definition Template.

The template section that appears outside the curly brackets corresponds to the model header, consisting of the following

- m*odel-name* is the name identifying the model. This name is internally used by NSLM to implicitly instantiate the complete model.

The template section that appears inside the curly brackets defines the actual structure and functionality of the model:

- *model-attribute-spec* defines the structure of the model in terms of primitive, object and module type variables. With the exception of not having any port instantiations, this section is similar to that in modules.
- *model-method-spec* defines the behavior of the model in terms of local functions or methods definitions, including simulation methods.

For example, the **ExtendedModel** instantiates the **ExtendedModule** module as shown in code segment 6.17.

```
nslModel ExtendedModel ()
{
   public ExtendedModule em(10,”p1”,’H’);
}
```

Code Segment 6.16
nslModel example.

ExtendedModel is implicitly instantiated by NSL in order to execute the model.

6.7 Summary

In this chapter we have presented the key concepts and constructs needed to build a NSL model. Since NSLM is built on top of C++ and Java, we discussed the parts of the NSLM language that is shared with these two native languages. We also discussed the basic NSLM types and how to build NSLM modules and models. Finally, we presented a section on how to build your own NSLM classes and what restrictions one might encounter when doing this. Although this chapter described the NSL modeling language in more detail than that presented in chapter 3, to fully grasp the language we recommend that Appendix I, The NSLM Methods be reviewed.

Notes

1. We explain how to expand many of the NSLM constructs in the NSLM Parser Guide for Java and C++ technical report that can be found on the NSL web site. These expansions usually vary between Java and C++.
2. We use the terms *primitive* and *native* interchangeably in this document, although many languages make a distinction between these two terms. Native types in NSLM reflect the native types present in both C++ and Java except for charString and boolean which are special cases. We also refer to all other types as NSLM types.
3. Indexing in similar to C++ array indexing except for the fact that no pointer arithmetic is allowed. Additionally, Java allows one array to be assigned to another, in which case the reference of the array is copied between the two variables and not the elements. If we were to assign one native array to another native array, we would need to use the verbatim keyword described below.
4. A boolean array can be converted to an **int**, **float**, or **double** array.
5. In the current NSL version only 0 dimension **NslString** objects are supported although future NSL version will include additional dimension types.
6. We avoid the use of the *new* operator as in C++ or Java since C++ returns a pointer and Java returns a reference.
7. See some of our current work on distributed simulation discussed Appendix III—NSLC Extensions.
8.
9. In addition to these two arguments, the NSLM compiler generates two more: “charString nslName” and “NslModule nslParent,” where nslName is the name of the instance being instantiated and nslParent is the instance of the parent module instantiating this module instance. The parser adds these parameters when the module is instantiated as well.
10. Currently the order of execution of modules defaults to their hierarchical order and the order in which they were defined in the module. However, we expect to offer more control in the scheduling order in the future.

11. We expect to provide in the future instantiation parameters for models, analogous to the *args* parameters in Java.