NodalCompML

Language Reference Manual

Oscar Batori (ojb2109)

Nina Berg (nb2555)

Victor Frenkel (vgf2103)

Venkata Yamajala (vsy2104)

Table of Contents

1 Introduction

NodalCompML is a language based on nodes processing data through their main compute functions. Nodes are connected to each other through inputs and outputs over which they forward data after having performed some computation on it. If a node receives new information on one of its inputs, it re-computes its outputs using the new data.

1.1 The Node

NCML allows program computation to be divided among multiple nodes. Each node has a unique name, as well as a list of arguments it can accept. See section 5.1 for details. Any node with an empty list of arguments is considered a start node.

The body of the node contains all computations related to that node including any local variable declarations, any local helper functions, as well as any forward statements. The main body of the node will be referred to as the node's compute function. Forward statements are used to pass data between nodes. See section 5.3 for more details on creating local helper functions. See section 6.7 for more details on forwarding.

A node will begin its computation when it has received values for all of its inputs. If multiple values are received for one input, the most recently received value will be used. Note that input values may arrive from multiple nodes.

2 Lexical Conventions

2.1 Comments

Comments begin with the `//` symbol and run until the end of the line.

2.2 Identifiers

Identifiers are case sensitive sequences of letters and digits, in which the first character of the sequence must always be a letter.

2.3 Keywords

The following identifiers are reserved:

`bool`	`break`	`char`

`continue`	`fun`	`return`
`double`	`node`	`string`
`else`	`if`	`while`
`for`	`int`	
`forward`	`interface`	

Note: `true` and `false` are not keywords, however it is illegal to use them as identifiers.

2.4 Constants and Literals

Integer constants are sequences of digits. Double constants consist of sequences of digits and a single decimal point `.` followed by an another optional sequence of digits.

String literals are sequences of letters and digits surrounded by double quotations ‟ ″. In order to use ‟ in a string it must be escaped by `\`. Boolean constants are either `true` or `false`.

2.5 Operators

The following symbols are reserved for operators:

`++`	`--`	`+=`
`-=`	`*=`	`/=`
`+`	`-`	`*`
`/`	`==`	`!=`
`<=`	`>=`	`>`
`<`	`%`	`||`
`&&`	`!`	`[]`

2.6 Punctuation

Statements and expressions do not have to be terminated by a semicolon. All that is required is if multiple expressions appear on a single line they be separated by commas.

3 Types

3.1 Primitive Types

NodalCompML supports the following basic types:

`char`: Can hold a single 16 bit character. The character can be a letter, digit, punctuation, or control character.

`int`: Signed 32-bit integer type. Can store any value in the range -2,147,483,648 to 2,147,483,647.

`double`: Double precision signed 64-bit floating point type.

`bool`: 1-bit data type with only two possible values, `true` or `false`.

3.2 Complex Types

`Type[]`: Describes an array, or collection of similar objects. Each element can be accessed by its location within the array. The size of the array must be declared when the array is first created. For example, `int[5]x` could be used to declare an array, `x`, of five integers. Upper limit on the size of the array is bound only by the amount of computing resources available.

`String`: Can hold a string of any size. Upper limit on the length of the string is bound only by the amount of computing resources available.

3.3 Type Conversion

Unary expressions of one type can be explicitly converted into another type by casting the expression as the desired type. Casting can be accomplished by preceding the expression with parenthesized name of the desired type. The syntax is:

(*type-specifier*) *expression*

Note that casting a more precise type into a less precise type may result in some data loss.

4 Expressions and Operators

4.1 Precedence and Associativity

The following table lists the precedence and associativity of NCML's operators. Operators in the same cell are of the same precedence and are associated as indicated. Rows are in order of decreasing precedence. Parentheses can be used to override precedence rules, as well as for clarity.

Operator	Description	Associativity
Primary Expression	Identifiers, Constants, Parenthesized Expressions	

`()`
`[]`
`++`
`--`
	Function calls
	Array element access
	Post Increment
	Post Decrement
		Left to right

`+ -`
`!`
`++`
`--`
`(type)`
	Unary plus and minus
	Logical NOT
	Pre increment
	Pre decrement
	Casting
		Right to Left

`*`
`/`
`%`
	Multiplication
	Division
	Modulus
		Left to Right

`+`
`-`
`+`
	Addition
	Subtraction
	String Concatenation
		Left to Right

`<`
`>`
`<=`
`>=`
	Relational Operators	Left to Right

`==`
`!=`
	Equality	Left to Right

`&&`	Logical AND	Left to Right
`||`	Logical OR	Left to Right

`= += -=`
`*= /=`
	Assignment	Left to Right

4.2 Primary Expressions

Primary expressions consist of identifiers, constants, string literals, or parenthesized expressions.

4.3 Function Calls

Function calls are made by calling the name of the function, followed by a parenthesized list of comma separated arguments. The argument list may be empty. The number of the arguments provided in the function call must match the number of arguments specified in the function definition. See section 5.3 for more details on function declarations.

4.4 Array References

The `[]` operator can be used to access individual elements within an array. For example `x[3]` will return a reference to element 3 in array x. Note that array indices begin at 0. The expression within the square brackets must evaluate to an integer.

4.5 Unary Operators

`+`, `-`: Results in the positive or negative value of the operand.

`++`, `--` : The value of the operand is incremented/decremented by one.

`!`: The result is the logical NOT of the operand.

`(type)`: Casting. See section 3.3 for more details.

4.6 Binary Arithmetic Operators

The following binary arithmetic operations are supported: multiplication `*`, division `/`, addition `+`, subtraction `-`, and modulus `%`.

Note that the + operator is also used for string concatenation.

4.7 Relational and Equality Operators

The following relational operators are supported: less (<), greater (>), less or equal (<=), greater or equal (>=), equal (==), and not equal (`!=`). The result of these expressions is of type bool, indicating when the relation is true or false.

4.8 Logical Operators

The logical AND operator, `&&`, returns true if both its operands evaluate to true, otherwise false is returned. The logical OR operator, `||`, returns true if either of its operators evaluates to true, otherwise false is returned. The logical NOT operator, `!`, is also supported. See section 4.5 for details.

4.9 Assignment

The = operator will replace the value of the left hand side with the evaluated expression on the right hand side. The complex assignment operators will perform the indicated binary operation with the right and left side values, and assign the result back to the left side. For example, `x += 1` will first evaluate x + 1 and then assign the result to `x`.

It is equivalent to writing `x = x + 1`.

5 Declarations

A declarator has exactly one decl-name specifying the identifier that is being declared. The syntax is as follows:

declarator:

decl-name

declarator [constant-expression[optional]]

decl-name:

identifier

See section 2.2 for more details on identifiers.

5.1 Node Declarations

A node specifies a blackbox model for a computation to be performed. The node specifier include a list of comma separated parameters to be received from other nodes. Input and output parameters can be given default values with the *param = <default value>* notation.

The syntax:

node-specifier:

`node` *declarator (argument-declaration-list[opt]) compound-statement*

argument-declaration-list:

argument-declaration-list, argument-declaration

argument-declaration

argument-declaration:

type-specifier declarator

type-specifier declarator = constant-expression

Details about the current NCML supported types can be found in section 4.

5.2 Interface Declarations

An interface is a node-like way for libraries to expose complete pieces of functionality to users. The interface declares necessary inputs and output variables. Every interface declaration must have both inputs and outputs. The user then tells the library interface where to send its output by using a forward statement. Only after telling the interface where to forward its outputs can the user forward data into the inputs of the interface. In this way, we can create reusable third party libraries using NCML.

The syntax is:

interface-specifier:

`interface` *declarator (argument-declaration-list) (argument-declaration-list)*

Where the first list is the list of inputs, and the second is the list of output locations.

See section 8 for an example using interfaces.

5.3 Function Declarations

Although the statements in the body of a node encapsulate a computation, it is often desirable to break up the computation into smaller helper functions that can be called from the node's compute function. Helper functions can be defined after the main compute function using the `fun` keyword. There may be no additional code within the node after the first helper function is declared, except additional helper functions.

The "type" of a function indicates the return type of the entire function. If the function returns nothing (void), the type of the function may be omitted.

The syntax for function declarations:

function-definition:

type-specifier[opt] `fun` *function-declarator function-body*

function-declarator:

declarator `(` *argument-declaration-list* `)`

function-body:

compound-statement

5.4 Variable Declarations

Variable declarations consist of a type specifier and the variable's declarator.

variable-declaration:

type-specifier declarator

6 Statements

6.1 If - Else Statement

NCML provides the standard if statement familiar from "C family" programming languages. Namely:

`if (` *expression* `)` *statement*

Where *statement* is executed only if *expression* evaluates to true (or a non-zero arithmetic value). Similarly, the usual if-else construction is provided:

`if (` *expression* `)` *statement1* `else` *statement2*

Where, as before, *statement1* is executed only if *expression* evaluates to true, and *statement2* is evaluated otherwise.

6.2 While Statement

Again NCML provides a standard while construction from the "C family," with the usual format:

`while (` *expression* `)` *statement*

Where *statement* is executed while *expression* evaluates to true. After every execution of *statement. expression* is evaluated to check if it is still true, in which case the loop continues, or false, in which case the loop terminates.

6.3 For Statement

Another classic "C family" construction is the for statement, which has the form

`for (` *expression1[opt]* `;` *expression2[opt]* `;` *expression3[opt]* `)` *statement*

Which is equivalent to

```
expression1
while ( expression2 ) {
    statement
    expression3
}
```

Where the first expression specifies initialization for the loop (this may not be necessary if the condition being tested in the second expression is defined within the correct scope, see below for scope rules). The second expression specifies the test that is conducted at the beginning of each iteration, and the third typically specifies the increment, performed after each iteration. All three expressions are optional, however, the `;`'s are mandatory.

6.4 Break Statement

As in "C-family" languages the statement `break` causes the termination of the smallest enclosing while or for statement, hence ending the loop. Control is passed to the statement immediately following the terminated statement.f

6.5 Continue Statement

Consistent with "C family" languages a continue statement is provided, where `continue` will pass control to the loop-continuation portion of the smallest enclosing while or for loop. More precisely, in the following statement

```
while ( expression1 ) {
        if ( expression2 ) {
                continue
        }
        statement
}
```

If *expression2* evaluates to true, then *statement* will not be executed and control will return to the top of the while loop (and *expression1* will be reevaluated). Otherwise, if *expression2* evaluates to false, statement will be executed.

6.6 Return Statement

As in "C family" languages a return statement of the form

```
return expression[opt]
```

Will return control back to the function caller. Return statements can be used from within a helper function to return control to the caller (which may be the main compute function or another helper). The main compute function cannot have a `return.` The optional *expression* can be used to specify a return value. The type of *expression* must match the return type of the function. If a function returns nothing (void), `return` may be omitted. In that case, control returns to the caller when the function has completed execution.

6.7 Forward Statement

Specific to NCML is the forward statement, which specifies where data from a given Node is forwarded. The forward statement must specify which node (or interface, see section 5.2) the data is to be forwarded to. The forward must also specify which argument within the node the data should be forwarded to. The syntax is as follows:

```
forward ( expression ) to ( declarator.declarator )
```

Where *declarator*`.`*declarator* is the name of the node followed by the input parameter. For example:

```
node myNode1() {
int x = 1
        forward x to myNode2.y
}
node myNode2(int y) {…}
```

In the example above, `myNode1` will forward the value of `x` to the `y` argument of `myNode2`. The type of *expression* must match the type of *declarator.declarator.*

Multiple forwarding statements may be combined. The order in which the forwarded values are listed determines where they are sent. For example, `forward (x, y) to (Node1.a, Node2.b)`. In this example, x will be forwarded to `Node1.a`, and y will be forwarded to `Node2.b`.

If multiple values are being forwarded to the same node, the following shortcut may be used: `forward (x, y) to node1.(a, b)`

7 Scope Rules

NCML is a lexically scoped language, i.e., a variable or function can only be referenced within the block where it was defined. Two identifiers may not share a name if they fall within the same scope. However, identifierss in different scopes may share a name. When an identifier is declared inside a block, any previous definitions of the identifier outside the block are suspended.

NCML does not permit variables or functions to be declared outside a Node or Interface.

Hence the following is not syntactically correct, assuming the correct Interfaces, and myNode2, definitions:

```
int x = 1;

node myNode () {
      forward (x) to (myNode1.y)
}
```

Whereas the following is correct, again assuming presence of correct definitions for myNode2, and the attendant Interfaces:

```
node myNode () {
      int x = 1
      forward (x) to (myNode1.y)
}
```

While one may argue that this limits code reuse, and to a degree it certainly does, this scoping, or rather the combination of lexical scoping, and restricted location of identifiers forces the programmer to use the nodal computation paradigm, and structure their program in terms of Nodes, rather than just a few Nodes, and a whole litany of helper functions. This is consistent with the mission of the language: it offers a new way to think about certain types of computation.

8 Example

Here is an example of an interface created in a library and used in a user program:

userprog.ncml

```
import mylib
import sys
```

```
node my_comp_start1 ( ) {
      image_filename = sys.argv[0]

      double x

      double y

      int[1000] in_image

      //various operations


      //define where output of interface goes

      forward (IComplicatedEffect.out_image) to (next_step.in_image)


      //kick off calculation by giving the interface its input.

      forward (in_image, x, y) to IComplicatedEffect.(in_image, param1,
param2)

}

node next_step (int[] in_image) {
      //do more stuff.
}
```

mylib.ncml

```
// declaring two interfaces here, IComplicatedEffect and IBlur


interface IComplicatedEffect (int[] in_image, double param1, double param2)
(int[] out_image) {
      forward (in_image, param1) to CEStart1.(image, param1)
      forward (in_image, param2) to CEStart2.(image, param2)
}

// a simpler interface that exposes a run through one node in this library
(blur).
interface IBlur(int[] image, float strength) (int[] out_image) {
      forward (image, strength) to Blur.(image, strength)
}


node Blur (int[] image, float strength) {
      int[] new_image = image
      //do stuff to image
      forward (new_image) to (IBlur.out_image)
}

node CEStart1 (int[] image, float param1) {
      int[] new_img
      //do stuff to new_img
      forward (new_img) to op1
}

node CEStart2 (int[] image, float param2) {
      int [] new_img
      //do stuff to new_img
      forward (new_img) to op1
}


node op1 (int[] image) {
      //do some stuff, forward some stuff
}

// a bunch of nodes here in the sequence that eventually forward to
CEFinalStep

node CEFinalStep(int[] image, float param3) {
      forward image to IComplicatedEffect.out_img
}
```