

# **System Builders Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)**

Thomas R. Kramer (thomas.kramer@nist.gov, phone 301-975-3518)  
John Horst (john.horst@nist.gov, phone 301-975-3430)

Intelligent Systems Division  
National Institute of Standards and Technology  
Technology Administration  
U.S. Department of Commerce  
Gaithersburg, Maryland 20899, USA

NISTIR 7715  
October 25, 2010

## **Disclaimer**

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

## **Acknowledgements**

Funding for the work described in this paper was provided to Catholic University by the National Institute of Standards and Technology under grant Number 70NANB9H9131.

# Table of Contents

1	Introduction. . . . .	1
1.1	Overview. . . . .	1
1.2	Arrangement of this Manual. . . . .	1
1.3	Use of Fonts . . . . .	2
1.4	Compiling the Tutorials . . . . .	2
2	C++ Classes Representing DMIS . . . . .	3
2.1	Overview. . . . .	3
2.2	Anatomy of the C++ Classes . . . . .	4
2.3	Naming C++ Classes and Attributes . . . . .	7
2.4	Automatically Generated C++ Attribute Names . . . . .	8
2.5	Automatically Generated C++ Class Names . . . . .	8
2.6	Using the C++ classes. . . . .	9
2.7	The isA Function . . . . .	11
2.8	Parse Tree . . . . .	12
3	Namespaces NDTs and NDTU . . . . .	13
3.1	Namespace NDTs . . . . .	13
3.2	Namespace NDTU . . . . .	14
4	The “makeBound” Tutorial Program . . . . .	15
4.1	What the Program Does . . . . .	15
4.2	How to Run the Program . . . . .	15
5	The “Generate” Tutorial Program. . . . .	17
5.1	What the Program Does . . . . .	17
5.2	How to Run the Program . . . . .	17
6	The “Analyze” Tutorial Program . . . . .	18
6.1	What the Program Does . . . . .	18
6.2	How to Run the Program . . . . .	18
	Appendix A Compiling Tutorials from Source Code in Windows . . . . .	20

# 1 Introduction

## 1.1 Overview

This is a system builders manual for the NIST DMIS Test Suite, version 2.2.1<sup>1</sup>. The purpose of this manual is to help system builders use software provided in the test suite for building systems that implement DMIS (the Dimensional Measuring Interface Standard).

The test suite and this manual were prepared at the National Institute of Standards and Technology (NIST). There are also a “Users Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)” and a “Maintainers Manual for the NIST DMIS Test Suite Version 2.2.1”. The users manual should be read (or scanned, at least) before reading this system builders manual because this manual assumes the reader understands things like “parser” that are explained in the users manual. Also, the users manual has information about DMIS 5.2. The test suite, which includes all three manuals, may be downloaded from

[http://www.isd.mel.nist.gov/projects/metrology\\_interoperability/dmis\\_test\\_suite.htm](http://www.isd.mel.nist.gov/projects/metrology_interoperability/dmis_test_suite.htm)

In addition, since the test suite is very large (so that prospective users may want to look at the manuals before deciding whether to download it), the manuals may be downloaded separately from the same site.

This manual includes descriptions of three sets of example source code. The first set, “makeBound”, focuses on the core of a DMIS generator implementation. The second set, “generate”, is a template for a DMIS generator implementation. The third set, “analyze”, is a template for a DMIS consumer application. The descriptions are given at the level of detail appropriate for someone who already knows C++ (including inheritance) and is comfortable writing C++ programs. The manual contains no exercises or problems for the reader to work. However, there are instructions for compiling the tutorial programs that could be followed as an exercise. Also, the reader may find it helpful to experiment by changing the source code of the tutorials, recompiling them, and running them.

## 1.2 Arrangement of this Manual

The remaining sections of this manual are:

- Section 2 (C++ Classes Representing DMIS) - describes the C++ classes that represent DMIS.
- Section 3 (Namespaces NDTs and NDTU) - describes how two C++ namespaces are used to help avoid naming conflicts.
- Section 4 (The “makeBound” Tutorial Program) - describes the C++ program named “makeBound” that shows how to use the C++ classes to generate one line of DMIS code.
- Section 5 (The “Generate” Tutorial Program) - describes the C++ program named “generate” that shows how to use the C++ classes to build a system that generates DMIS input files.
- Section 6 (The “Analyze” Tutorial Program) - describes the C++ program named “analyze” that shows how to use the parser and the C++ classes to build a system that reads DMIS input files and processes them.

---

1. In the remainder of this manual “the test suite” means the NIST DMIS Test Suite, version 2.2.1.

## 1.3 Use of Fonts

In this manual:

- DMIS code is shown in *this font*.
- File and directory names are shown in *this font*.
- C++ code is shown in *this font*.
- Commands typed in a command window are shown in **this font**.
- DEBNF<sup>1</sup> code is shown in *this font*.

## 1.4 Compiling the Tutorials

The tutorial programs may be compiled using any modern C++ compiler. The tutorials are already compiled for Linux, SunOS, and Windows, so it is not necessary to recompile them unless they will not run on your system. If you want to recompile them on your system, continue reading this section.

### 1.4.1 Linux

For Linux, edit the Makefile in tutorials/linuxSun so that LINCOMPILE and LINLINK are set to point to your C++ compiler. Then the tutorials can be recompiled from the tutorials/linuxSun directory with the following commands.

***make binLinux/makeBound***

***make binLinux/analyze***

***make binLinux/generate***

### 1.4.2 Sun

For SunOS, edit the Makefile in tutorials/linuxSun so that SUNCOMPILE and SUNLINK are set to point to your C++ compiler. Then the tutorials can be recompiled from the tutorials/linuxSun directory with the following commands.

***make binSun/makeBound***

***make binSun/analyze***

***make binSun/generate***

### 1.4.3 Windows

For Windows, the tutorials (and all other C++ code) have been compiled using the Microsoft Visual C++ 2008 Express Edition, which may be downloaded from <http://www.microsoft.com/express/vc> and used with no charge. This compiler must be run using its graphical user interface.

To rebuild an already-built executable:

- Start Visual C++.
- From the File menu, select Open.
- In the Open Project popup window that appears, use the browser to choose the project you want. Projects have a “.sln” suffix (analyze.sln, for example). Then press the Open button. The popup will disappear.
- From the Build menu, select Rebuild Solution.
- Select Save All from the File menu, then select Exit from the File menu.

---

1. DMIS Extended Backus-Naur Form

Instructions for compiling the tutorials in Windows starting from source code are given in Appendix A. If all you want do is change existing source code and then recompile, the instructions above should work.

## 2 C++ Classes Representing DMIS

### 2.1 Overview

The C++ classes represent full DMIS. They may be found in the `dmis.hh` and `dmis.cc` files in the `utilityComponents/linuxSun/source` directory or in the `dmis.h` and `dmis.cpp` files in the `utilityComponents/windows/source` directory. There are a lot (1993) of C++ classes for DMIS.

If you are building a DMIS generator, you use the classes by populating them in a program which eventually calls a single `printSelf` function to generate a file of DMIS code. If you are building a DMIS consumer, you use the classes in a program that, near the beginning, calls a single `parse` function to read in a file of DMIS code. The parsing automatically builds a parse tree consisting of instances of the C++ classes. Then the rest of your program traverses and processes the parse tree to do whatever you want with it.

Section 9.5 of the users manual for the test suite describes DEBNF in detail. Briefly:

- A DEBNF file is a formal description of part or all of the DMIS language.
- A DEBNF file is a list of productions.
- A production sets a name to be equivalent to any of a list of definitions.
- Each definition is a list of expressions<sup>1</sup>.
- An expression is a name (of a fixed symbol or a production), or a single character, or an optional list of expressions (or a couple other things less frequently).

The C++ classes for DMIS were built automatically from the DEBNF file for DMIS. All the C++ names of classes and attributes are derived from names given in the DEBNF either as names of productions or as comments. Section 2.3, Section 2.4, and Section 2.5 give details on how classes and attributes are named.

The rules for determining whether a class will be defined to represent something are simple.

First, a class is defined for every production in the DEBNF that does not

- define a list,
- give a dummy definition for a terminal, or
- give the spelling of a token name.

Second, if a DEBNF production has two or more definitions, an additional class is defined for each definition, and the class for the production is the parent of each additional class.

The DEBNF tends to be in a deep hierarchy with short definitions rather than in a shallow hierarchy with long definitions. Since the C++ class hierarchy follows the DEBNF hierarchy, it is deep, too.

---

1. In formal DEBNF, the expressions are separated by commas. In this manual, the separating commas are omitted to make the DEBNF easier to read.

## 2.2 Anatomy of the C++ Classes

When a class is constructed from a production that has more than one definition, it has the form shown in Figure 1. There is a single class, **dmisCppClassBase**, from which all other classes are derived. It exists in order to introduce the virtual function **printSelf**. It also has the form shown in Figure 1 except that it does not have the **public parentClass** line.

```
class aClass :
    public parentClass
{
public:
    aClass();                // constructor with no arguments
    ~aClass();               // destructor
    void printSelf() = 0;     // virtual printSelf method
};
```

**Figure 1. Archetype C++ Parent Class**

When a class is constructed from a production that has a single definition, the class has the form shown in Figure 2. The names in **bold type** in Figure 1 and Figure 2 will change from class to class. All other characters in those figures outside of comments are the actual characters that are used.

```
class aClass :
    public parentClass
{
public:
    aClass();                // constructor with no arguments
    aClass(
        firstType * firstTypeIn,    // constructor with arguments
        lastType * lastTypeIn);    // first argument to constructor
    ~aClass();               // last argument to constructor
    void printSelf();        // destructor
    firstType * get_firstType();    // printSelf method
    void set_firstType(firstType * firstTypeIn); // get function for first attribute
    lastType * get_lastType();    // set function for first attribute
    void set_lastType(lastType * lastTypeIn); // get function for last attribute
private:
    firstType * a_firstType;    // set function for last attribute
    lastType * a_lastType;    // first attribute
    // last attribute
};
```

**Figure 2. Archetype C++ Class**

The *printSelf* function declared in Figure 2 is implemented for all such classes in *dmis.cc* (for Windows, *dmis.cpp*). The *printSelf* functions print the DMIS code represented by the classes. The *printSelf* functions are powerful because the *printSelf* function for each class, in addition to printing whatever DMIS is needed for the class itself, calls the *printSelf* functions for the attributes of the class. Thus, an entire DMIS file can be printed by a single call to *inputFile::printSelf()*. The “generate” tutorial provides an example of this.

There is only one function that prints DMIS that is not a *printSelf* function. That is the *printDouble* function. It prints a *double* with the default number of decimal places (six, usually) but it suppresses trailing zeros. For example, it prints 3.65 rather than 3.650000.

The constructor that takes no arguments does not set any attribute values.

The constructor that takes arguments takes one argument for each attribute, and the type of that argument is the same type as the type of the attribute. The constructor sets the value of each attribute to the value given in the arguments.

The destructor does not free any memory for attributes that may have been allocated.

The archetype in Figure 2 is shown with two attributes, but in general, there may be zero to many attributes. For each attribute there is:

- a private attribute,
- a public *get\_attribute* function that returns the value of the attribute,
- a public *set\_attribute* function that sets the value of the attribute,
- an argument to the constructor that takes arguments.

The names used for arguments in the constructor and the set functions are formed by adding the suffix *ln* to the base names of the attributes. The names used for attributes are formed by adding the prefix *a\_* to the base names of the attributes.

When there are no attributes, there is no constructor that takes arguments, and there are no *get\_attribute* or *set\_attribute* functions.

Only one type of aggregate is used; that is list. Every list is of the form *std::list<something>*. All of the standard C++ list manipulation functions will work with every list.

The value of every attribute is a pointer of some sort except when an attribute of a class is a *bool* or a non-optional *int* or *double*. When an attribute is optional and its type is *int* or *double*, the value of the attribute is a pointer.

Only those items that can differ between instances of a class are represented as attributes of the class. Items such as statement names and commas that are always the same in every instance of a class are not represented as attributes of the class. Figure 3 shows the definition of the *boundStm* class as given in *dmis.hh*. As shown in Figure 3, a *boundStm* has the DEBNF definition *BOUND* *''' boundMinor #*. The only thing in a *boundStm* that varies between instances is the *boundMinor*, so only the *boundMinor* is represented in the *boundStm* class. It is represented by the attribute *a\_boundMinor*, which is a pointer to a *boundMinor*. The class name *boundStm* is obtained by using the production name *boundStm* from the DEBNF. The production name is not shown on Figure 3; only its definition is shown.



```
/* boundStm
```

This is a class for the single definition of boundStm. It represents the following items:

```
BOUND '/' boundMinor #
```

```
*/
```

```
class boundStm :
    public dmisFreeStatement,
    public dmisStatement
{
public:
    boundStm();
    boundStm(
        boundMinor * boundMinorIn);
    ~boundStm();
    void printSelf();
    boundMinor * get_boundMinor();
    void set_boundMinor(boundMinor * boundMinorIn);
private:
    boundMinor * a_boundMinor;
};
```

**Figure 3. boundStm C++ Class**

When part of a DMIS statement is optional and contains items that can differ between instances of the statement, there is an attribute (which is a pointer) for each of those items. If an instance of the statement does not include the optional part, then in the class instance representing the statement, the pointers for the items in the optional part are null pointers (i.e. the value of the pointer is 0).

When part of a DMIS statement is optional but contains only items such as keywords and commas that do not differ between instances of the statement that have the optional part, there is a boolean attribute for the optional part. In class definitions, this is a *bool* with the usual allowed values of *true* and *false*.

The *callMacro* class shown in Figure 4 provides an example of the points in the preceding two paragraphs. The optional [ ' , ' CHARSTRING ] at the end of the DEBNF text represents a string consisting of the arguments to the macro being called. If a DMIS CALL statement has a comma and string such as , ' 3 , 2.5 ' at the end, then the value of the *a\_string* attribute of the instance of the *callMacro* class representing the statement will be a pointer to the string "3, 2.5". If not, the value will be null. If the statement has EXTERN, DMIS, at the beginning, then the value of the *has\_EXTERN* attribute of the class instance will be *true*. If not, the value will be *false*.

```

/* callMacro

This is a class for the single definition of callMacro. It represents the following items:
[EXTERN ' , ' DMIS ' , ' ] mLabel [ ' , ' CHARSTRING]
*/

class callMacro :
    public callMinor
{
public:
    callMacro();
    callMacro(
        bool has_EXTERNIn,
        mLabel * mLabelIn,
        char * stringIn);
    ~callMacro();
    void printSelf();
    bool get_has_EXTERN();
    void set_has_EXTERN(bool has_EXTERNIn);
    mLabel * get_mLabel();
    void set_mLabel(mLabel * mLabelIn);
    char * get_string();
    void set_string(char * stringIn);
private:
    bool has_EXTERN;
    mLabel * a_mLabel;
    char * a_string;
};

```

**Figure 4. callMacro C++ Class**

### 2.3 Naming C++ Classes and Attributes

The method of naming attributes is intended to make the meaning of the attributes clear to anyone who knows DMIS. When the names of productions used in the DEBNF file for DMIS convey what they mean intuitively, the production names are used to make C++ names. In many cases, the production names or names derived from them are not very good. In the last release of the test suite, it was necessary to live with bad names.

This release, however, implements a method of ensuring that good names are used. To assign a base name to an attribute, an EBNF comment of the form (*\*A=name\**) may be inserted immediately after any expression in an EBNF definition. The C++ generator then assigns the name to the attribute derived from the expression. About 800 attribute names have been assigned that way. To assign a name to a class, an EBNF comment of the form (*\*C=name\**) may be inserted immediately after an EBNF definition. The C++ generator then assigns the name to the

class derived from the definition. About 270 class names have been assigned that way. If a name has not been inserted manually, then the automatic methods described in Section 2.4 and Section 2.5 are used.

## 2.4 Automatically Generated C++ Attribute Names

The names of automatically generated C++ attributes are formed by concatenating the prefix *a\_* with the type of the data. For example, in Figure 4, the attribute name *a\_mLabel* is used where the data type is *mLabel*.<sup>1</sup>

When the data type is *char \**, the attribute name is *a\_string*, since *a\_char* sounds like a single char.

In the case of an attribute whose value is a list, the name of the attribute is taken from the name of the list in DEBNF, with the *a\_* prefix added as described above. In most cases this means that the name (before the prefix or suffix is added) is made by concatenating the type of thing listed with *List*. For example, an attribute of type *std::list<dmisltem \*>* has the name *a\_dmisltemList*. In some cases, the DEBNF name was not formed by concatenating, so the attribute name is irregular. For example, a *boundFeat* has an attribute named *a\_featureList* which is a list of *featureLabel*, not a list of *feature*.

When the data type is *bool*, the attribute name is made by concatenating the prefix *has\_* with the first name in the optional items being represented. For example, in Figure 4, the attribute that indicates whether *EXTERN*, *DMIS*, is used is named *has\_EXTERN*.

## 2.5 Automatically Generated C++ Class Names

The automatically generated name of each class corresponding to an entire production is the same as the name of the production. When the production has only one definition, only one class is defined.

When there are two or more definitions for a production, an additional class is defined for each definition. If possible, the name for the class for each definition is given the form *productionName\_itemName*, where *itemName* is the name of one of the expressions in the definition.

That form is possible when any of the following three conditions holds.

(1) Every definition has exactly one expression and each of those expressions has a name (not all expressions have names). In this case, *itemName* is the name of the expression.

For example, if the DEBNF is

```
callType = WAIT | CONT | ATTACH ;
```

then the class names will be *callType\_WAIT*, *callType\_CONT*, and *callType\_ATTACH*.

(2) Every definition starts with a keyword or a nonterminal and no two definitions start with the same thing. In this case *itemName* is the name of the first expression in the definition.

---

1. If there are two or more occurrences of the same type of data in a class, this method does not work. An automatic method has been implemented for these cases that forms the name by adding *\_1*, *\_2*, etc. Since this always makes bad names, however, in every case where the method would be used, a name has been assigned using a comment, so no names of that sort will be found in the C++.

For example, if the DEBNF is

```
sensorMltprbItem = stringVal ',' sensorProbeGeometry
                  / intVal ',' sensorProbeGeometry ;
```

then the class names will be *sensorMltprbItem\_stringVal* and *sensorMltprbItem\_intVal*.

(3) Every definition is identical, except for one term that is either a keyword or a nonterminal. In this case *itemName* is the name of the distinguishing term.

For example, if the DEBNF is

```
constArc = ARC ',' fLabel ',' bfConst
          / ARC ',' fLabel ',' projctConst
          / ARC ',' fLabel ',' trConst ;
```

then the class names will be *constArc\_bfConst*, *constArc\_projctConst*, and *constArc\_trConst*.

In all cases in which none of the three conditions above holds, a class name has been assigned in the EBNF file, and that is used. If none of the above conditions held and the EBNF file did not provide a class name, then an automatic method using suffixes *\_1*, *\_2*, etc. for constructing class names would be called, and it would make a poor name; in this version of the test suite, that never happens.

## 2.6 Using the C++ classes

You need to understand C++ and DMIS in order to use the C++ classes that represent DMIS. Once you understand as much of DMIS as you plan to implement, using the C++ classes is not difficult. To find the class or classes required to deal with a particular DMIS statement, however, two documents are needed: (1) a copy of the DMIS 5.2 standard (electronic or paper) and (2) an electronic copy of the header file defining the classes. Since there are a lot of classes (1993 for full DMIS, covering over 50,000 lines in the header file), looking through the header file manually will not work.

The quickest way to find the classes needed to deal with a particular DMIS statement is to begin by searching the header file for *class statementStm*, where *statement* is the name of the statement in lower case letters. For example, if you want to find the classes for the BOUND statement, search for *class boundStm*. That will get you very quickly to the class for the statement. In a few cases, that is all you need to do. In most cases, however, the class for the statement has attributes that are other classes, and you will need to look at those classes, and they, in turn, may have attributes, and so on through perhaps five or six levels.

For example, instances of six classes are used in constructing the DMIS BOUND statement *BOUND/F(f1), F(f2), F(f3)*. They are: *boundStm*, *boundFeat*, *fLabel*, *labelNameCon*, *labelNameConst*, and *featureLabel*. The C++ code (from *dmis.hh*) for the first two of these (and *boundMinor*) is shown in Figure 5.

```

/* boundStm - represents: BOUND '/' boundMinor # */
class boundStm :
    public dmisFreeStatement,
    public dmisStatement
{public:
    boundStm();
    boundStm(
        boundMinor * boundMinorIn);
    ~boundStm();
    void printSelf();
    boundMinor * get_boundMinor();
    void set_boundMinor(boundMinor * boundMinorIn);
private:
    boundMinor * a_boundMinor; };

/* boundMinor - This is a parent class. */
class boundMinor :
    public dmisCppBase
{public:
    boundMinor();
    ~boundMinor();
    void printSelf() = 0; };

/* boundFeat - represents: fLabel ',' featureList */
class boundFeat :
    public boundMinor
{public:
    boundFeat();
    boundFeat(
        fLabel * fLabelIn,
        std::list<featureLabel *> * featureListIn);
    ~boundFeat();
    void printSelf();
    fLabel * get_fLabel();
    void set_fLabel(fLabel * fLabelIn);
    std::list<featureLabel *> * get_featureList();
    void set_featureList(std::list<featureLabel *> * featureListIn);
private:
    fLabel * a_fLabel;
    std::list<featureLabel *> * a_featureList;};

```

**Figure 5. C++ Some Classes for BOUND/F(f1),F(f2),F(f3)**

The *boundStm* class has one attribute, *a\_boundMinor*, and its value is a *boundFeat*. The C++ code and the attribute name show that the value of the attribute *a\_boundMinor* is a *boundMinor*. *boundMinor*, however, is a parent type that is not intended to be instantiated. *boundFeat* is the child class of *boundMinor* that matches the DMIS code we want to build (since BOUND is followed by  $F(f1)$ ), so we use an instance of *boundFeat*.

The *boundFeat* instance has two attributes, *a\_fLabel* and *a\_featureList*. The value of *a\_fLabel* is an instance of *fLabel*. The value of *a\_featureList* is a *std::list<featureLabel \*>* which, in this case, has two elements, both of which are *fLabels*.

Continuing the analysis through *fLabel*, *labelNameCon*, and *labelNameConst* is left to the reader.

## 2.7 The isA Function

The *isA* function is provided to get run-time information about the type of an object (i.e., an instance of a class). This is useful for dealing with a parse tree. Frequently, in analyzing a parse tree, you will know that an object is of some parent type and will need to know what child type it is. That's where you use *isA*. The function takes two arguments: an object, and a type, so that a call has the form *isA(object, type)*.

For example, suppose in your application you have an instance of a *callRoutine*. One of the attributes is *a\_callType*, which is (of course) a *callType*. You need to determine whether it is a *callType\_WAIT*, a *callType\_CONT*, or a *callType\_ATTACH* because your application will take different actions according to what it is. So you write (completely ordinary) if, else if, ... else code using the *isA* function as your test:

```
if (isA(a_callType, callType_WAIT))
    action1;
else if (isA(a_callType, callType_CONT))
    action2;
else if (isA(a_callType, callType_ATTACH))
    action3;
```

The *analyzeItems* function in the “analyze” tutorial program provides another example of this kind of code.

The *isA* function is not a normal function. It could not be a normal function since one of the arguments is a type. Instead, *isA* is the following compiler macro

```
#define isA(a,b) dynamic_cast<b *>(a)
```

This uses C++'s *dynamic\_cast* construct. The *dynamic\_cast* construct is the standard C++ method of casting an object known to be polymorphic. Normal C style casts do not work on polymorphic objects. *Dynamic\_cast* does not work on an object that is not polymorphic, but every instance of any of the C++ classes for DMIS is polymorphic, so *dynamic\_cast* will always work. *Dynamic\_cast* returns a pointer to an object of the type being tested if the object being tested is of the type being tested and null if not.

Often, once you have determined that an object is of a particular type, you will want to cast it into that type. To do that, call *dynamic\_cast* explicitly as shown in Figure 6.

```

int foo(callType * callType1)
{
    callType_WAIT * callType2;
    if (isA(callType1, callType_WAIT))
    {
        callType2 = dynamic_cast<callType_WAIT *>(callType1);
        doSomething(callType2);
    }
}

```

**Figure 6. Using dynamic\_cast with isA**

You can, of course, combine the assignment and the test, in which case you do not use `isA` at all. This is shorter but a little obscure, as shown in Figure 7.

```

int foo(callType * callType1)
{
    callType_WAIT * callType2;
    if ((callType2 = dynamic_cast<callType_WAIT *>(callType1));)
    {
        doSomething(callType2);
    }
}

```

**Figure 7. Using dynamic\_cast without isA**

## 2.8 Parse Tree

When the *yyvsparse* function runs in the parser, it parses a DMIS file and builds a parse tree named *tree* that represents the file. In C++ terms, the parse tree is an instance of the *inputFile* class. An *inputFile* has three attributes, as follows:

```

dmisFirstStatement * a_dmisFirstStatement;
std::list<dmisItem *> * a_dmisItemList;
endfilStm * a_endfilStm;

```

The list of *dmisItems* in the middle of the parse tree can be conveniently examined one at a time by a *for* loop that iterates using a standard list *iterator* in a function of the sort shown in Figure 8.

```

void doltems(std::list<dmisItem *> * items)
{
    std::list<dmisItem *>::iterator iter;
    for (iter = items->begin(); iter != items->end(); iter++)
    {
        if (isA((*iter), type1))
            ...
        else if (isA((*iter), type2))
            ...
    }
}

```

**Figure 8. For Loop on dmisItem**

The same sort of *iterator* and *for* loop can be used to examine any list. Just change the type of thing listed.

If there are more than 100 different possibilities (*dmisFreeStatement* has 179), the Windows C++ compiler may be unable to handle the code. In this case, the

```
if (A) {do_a();} else if (B) {do_b();} ... else if (Z) {do_z();}
```

construct may be replaced with

```
if (A) {do_a(); return;} if (B) {do_b(); return;} ... if (Z) {do_z(); return;}
```

The “analyze” tutorial described in Section 6 uses an *iterator* and a *for* loop of the sort shown above.

### 3 Namespaces NDTs and NDTU

In order to avoid name conflicts when you use test suite source code and libraries in your C++ programs, all of the test suite code is in one of two namespaces NDTs or NDTU. Namespaces are the standard C++ method of modularizing code.

#### 3.1 Namespace NDTs

To use the C++ classes for DMIS in the test suite (plus access functions, parser, and printer), the NDTs namespace has to be accessed. There are two ways to do this. Both methods start by putting `#include “dmis.hh”` (*dmis.h* for Windows) near the beginning of your program, as shown in Figure 11. That will enable you to use all the DMIS classes, including their constructors and access functions.

In the first method of using the NDTs code and library, you put the prefix *NDTs::* in your code before every NDTs name you use. In this method, if you want to use the DMIS parser, you will also need to put into your file the namespace declaration that is shown in Figure 9. This must come after the `#include dmis.hh` line. The first method is used in the `generate.cc` and `analyze.cc` tutorials.

In the second method of using the NDTs code and library, you put a *using namespace NDTs;*



line in your code (after the `#include dmis.hh` line), as shown in Figure 11. This puts all the NDTs names in your namespace, so if you use this method, you may get name conflicts. If you do, you must either change your names or use the first method. If you use the second method and you want to use the DMIS parser, you will have to include extern declarations in your code similar to the ones in Figure 9, but without the `namespace{ }` around them.

```
namespace NDTs {  
    extern FILE * yyin;  
    extern int numErrors;  
    extern int numWarnings;  
    extern inputFile * tree;  
  
    void preprocess(char * fileNameIn);  
    void resetParser();  
    int yyparse();  
}
```

**Figure 9. Namespace Declaration for Using Parser**

### 3.2 Namespace NDTU

The test suite utilities and their helper files are all in namespace NDTU. The utilities can be called from your C++ program by including a namespace NDTU declaration in your program. The object file for the utility you want to use and the library must be linked with the object file for your code. The very short driver files (`dmisParserDriver.cc`, `dmisConformanceCheckerDriver.cc`, `dmisConformanceTesterDriver.cc`, and `dmisConformanceRecorderDriver.cc`) in `utilityComponents/linuxSun/source` all provide examples of how to do this. The entire `dmisConformanceTesterDriver.cc` file is shown in Figure 10. The only thing you need to put in the namespace declaration is the function declaration for the utility you want to use.

```
namespace NDTU {  
    int testDmis(int argc, char * argv[]);  
}  
  
int main(int argc, char * argv[])  
{  
    return NDTU::testDmis(argc, argv);  
}
```

**Figure 10. dmisConformanceTesterDriver.cc**

## 4 The “makeBound” Tutorial Program

### 4.1 What the Program Does

The makeBound tutorial program shown in Figure 11 generates and prints the DMIS statement `BOUND/F(f1),F(f2),F(f3)`. The code may also be found in `tutorials/linuxSun/source/makeBound.cc` and `tutorials\windows\source\makeBound.cpp`.

The program defines a *makeBound2* function that returns a pointer to a *boundStm* and defines a *main* function that calls *makeBound2*. When the program is run, it prints `BOUND/F(f1),F(f2),F(f3)`. The general approach of the function is to make a tree of constructors. There is no constructor for a populated list, however, so the list is populated using *push\_back*, the standard C++ function for adding items at the end of a list.

For Linux and Sun, the files `dmis.hh` and `dmis.a` are used in compiling the program. For Windows, the files `dmis.h` and `dmis.lib` are used.

### 4.2 How to Run the Program

#### 4.2.1 Linux

In a Linux terminal window, get into the `tutorials/linuxSun` directory, and give the command:

***binLinux/makeBound***

#### 4.2.2 Sun

In a Sun terminal window, get into the `tutorials/linuxSun` directory, and give the command:

***binSun/makeBound***

#### 4.2.3 Windows

In a Windows command window, get into the `tutorials\windows\makeBound` directory, and give the command:

***Debug\makeBound***

```
#include "dmis.hh"

using namespace NDTs;

boundStm * makeBound2(
    char * boundLabel,
    char * featLabel1,
    char * featLabel2)
{
    std::list<featureLabel *> * featureLabels;

    featureLabels = new std::list<featureLabel *>;
    featureLabels->push_back(new fLabel
        (new labelNameCon
            (new labelNameConst(featLabel1))));
    featureLabels->push_back(new fLabel
        (new labelNameCon
            (new labelNameConst(featLabel2))));
    return new boundStm
        (new boundFeat
            (new fLabel
                (new labelNameCon
                    (new labelNameConst(boundLabel))),
                featureLabels));
}

int main()
{
    makeBound2("f1", "f2", "f3")->printSelf();
    return 0;
}
```

**Figure 11. C++ Program to Make BOUND/F(f1),F(f2),F(f3)**

## 5 The “Generate” Tutorial Program

### 5.1 What the Program Does

The “generate” tutorial program builds a specific DMIS input file. The source code is in `tutorials/linuxSun/source/generate.cc` and in `tutorials\windows\source\generate.cpp`. This program illustrates how to use the C++ classes in a program that generates DMIS input files. The `generate.cc` file contains good in-line documentation.

The general approach is:

- Define helper functions for building instances of frequently used classes and classes with a lot of substructure.
- Call the helper functions or constructors repeatedly to build a list of DMIS statements.
- Sandwich the list of DMIS statements between a *dmismnStm* and an *endfilStm* to make an *inputFile* named *theFile*.
- Call *theFile->printSelf()* to print the DMIS input file.

The helper functions are similar to the *makeBound2* function in Figure 11. They are:

- *makeCartPtmeas* - takes six doubles representing a point and a surface normal and returns a pointer to a *ptmeasStm* with those values.
- *makeLabel* - takes a string containing the name for a label and returns a pointer to a *labelNameCon* with that name.
- *makePtGoto* - takes three doubles representing a point and returns a pointer to a *gotoStm* saying to go to that point.

For Linux and Sun, the files `dmis.hh` and `dmis.a` are used in compiling the program. For Windows, the files `dmis.h` and `dmis.lib` are used.

### 5.2 How to Run the Program

#### 5.2.1 Linux

In a Linux terminal window, get into the `tutorials/linuxSun` directory, and give the command:

***binLinux/generate***

#### 5.2.2 Sun

In a Sun terminal window, get into the `tutorials/linuxSun` directory, and give the command:

***binSun/generate***

#### 5.2.3 Windows

In a Windows command window, get into the `tutorials\windows\generate` directory, and give the command:

***Debug\generate***

## 6 The “Analyze” Tutorial Program

### 6.1 What the Program Does

The “analyze” tutorial program shows how the parser and the C++ classes for DMIS can be used in a system that consumes DMIS input files.

The “analyze” program counts the total number of times each of several nominal feature types is defined in a set of DMIS input files. This is one of the simplest things a DMIS consumer could do. The kind of DMIS consumer program that is most interesting and useful would be a DMIS executor, but even the simplest executor is too complex for a tutorial.

The source code for the program is in `tutorials/linuxSun/source/analyze.cc` and in `tutorials\windows\source\analyze.cpp`. The source code for the “analyze” program has only five functions, but the program also calls three functions defined in `dmisYACC.cc`.

1. The `main` function calls *analyzeManyFiles* to count the number of instances of each feature type used in a set of DMIS input files, and then calls *reportResults* to print the results.
2. The *analyzeManyFiles* function takes a string argument, *fileNameFile*, and expects it to be the name of a file that contains the names of a number of DMIS input files. For each file listed in the *fileNameFile*, *analyzeManyFiles* calls *analyzeOneFile*.
3. The *analyzeOneFile* function:
  - preprocesses the file whose name is *fileName*.
  - opens the preprocessed file and sets *yyin* to the opened file.
  - exits if the preprocessed file did not open.
  - calls *yparse*; this parses the preprocessed file and builds a parse tree.
  - closes *yyin*.
  - deletes the preprocessed file.
  - reports the number of errors and warnings.
  - calls *analyzeItems* if there were no errors or warnings and there are items to analyze.
  - resets the parser so it is ready to parse another file.

Almost every DMIS consumer program that uses the parser and C++ classes would include all the steps in the *analyzeOneFile* function, except that the *analyzeItems* function would be replaced.

4. The *analyzeItems* function looks through the *dmisItem* which was built when a DMIS input file was parsed and adds 1 to the number of instances of a type of feature whenever that type of feature is found among the *dmisItems* being analyzed.
5. The *reportResults* function prints the total number of times each feature type was found in the DMIS input files that were examined.

### 6.2 How to Run the Program

The `runSomeFull` file contains a list of the names of DMIS input files, so “runSomeFull” may be used as a command argument with the “analyze” program. You can substitute the name of some other list of DMIS input file names, but be sure the file names are complete relative or absolute path names.

### 6.2.1 Linux

In a Linux terminal window, get into the tutorials/linuxSun directory, and give the command:

***binLinux/analyze runSomeFull***

### 6.2.2 Sun

In a Sun terminal window, get into the tutorials/linuxSun directory, and give the command:

***binSun/analyze runSomeFull***

### 6.2.3 Windows

In a Windows command window, get into the tutorials\windows\analyze directory, and give the command:

***Debug\analyze runSomeFull***

## Appendix A Compiling Tutorials from Source Code in Windows

This appendix gives instructions for making the executable “analyze” from source code using the Microsoft Visual C++ 2008 Express Edition. If you are using some other version of Visual C++, these exact instructions are not likely to work, but they may be helpful hints.

The easy way to compile the executable “analyze” is described in Section 1.4.3. The instructions in this appendix are intended to be used only if the easy way does not work. These instructions assume that the **analyze** subdirectory of the **tutorials\windows** directory does not yet exist. So, if you want to try these instructions, first delete or rename the **analyze** subdirectory of **tutorials\windows**.

These instructions also work for the executable “generate”. Just substitute

- “generate” for “analyze”.

These instructions also work for the executable “makeBound”. Just substitute:

- “makeBound” for “analyze”

1. Start Visual C++. If it is already running, shut it down and restart it.
2. From the **File** menu, select **New** and then **Project**. This brings up a popup with two large boxes on top, and three long thin boxes on the bottom, with a check box after the last one.
3. In the top left (**Project types**) box, select **Win32**.
4. In the top right (**Templates**) box, select **Win32 Console Application**.
5. In the bottom boxes put:  
**Name** - analyze  
**Location** - <NDTS>\tutorials\windows\  
    where <NDTS> is the full path to the test suite, for example:  
    R:\proj\dmis\kramer\NistDmisTestSuite2.2.1  
**Solution Name** - analyze  
**Create directory for solution** - leave checked  
Then press **OK**.
6. In the popup that appears, press **Next** (not **Finish**).
7. This brings up a popup labeled **Application Settings**.  
    Under **Application Type**, select **Console Application**.  
    Under **Additional Options**, first uncheck **Precompiled Header**, then check **Empty Project**.  
    Then press **Finish**.  
    This puts control back into the main Visual C++ window.
8. To get the project to use the source code, in the **Project** menu of the main window, select **Add Existing Item**. This brings up a file browser window. It may be necessary to select **Add Existing Item** twice, since only one item at a time can be added.

From the <NDTS>\utilityComponents\windows\source directory, select the following

source code file, and then press Add:

dmis.h

From the <NDTS>\tutorials\windows\source directory, select the following source code file, and then press Add:

analyze.cpp

Visual C++ will appear to put the files in a location shown in the Solution Explorer hierarchy window on the left of the main window. This is a project hierarchy, not a directory hierarchy (although it looks like a directory hierarchy). If the source code is put in the wrong place, it can be dragged up or down the hierarchy into the right place. Header files go in the fake HeaderFiles directory, and .cpp files go in the fake SourceFiles directory.

9. To get the project to use the dmis library, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window.

From the <NDTS>\utilityComponents\windows\dmisClasses\Debug directory, select dmis.lib and then press Add.

When you add dmis.lib, Visual C++ will display a popup window asking if you want to create a rule for making dmis.lib.

Press the No button.

In the Solution Explorer window, dmis.lib goes directly into analyze, not in any fake directory.

10. Even though Visual C++ knows exactly where the dmis.h file is (and will display it if you double click on it in the Solution Explorer window), Visual C++ does not find dmis.h (which is #include'd by analyze.cpp) when it is compiling analyze.cpp unless you do the following.

From the Project menu of the main window, select Properties.

This will bring up a popup window with a box on the left side containing a hierarchy of properties. Expand Configuration Properties. Then expand C/C++. Then select Command Line. A box labeled Additional options will appear at the lower right of the popup. In that box, enter:

```
/I ..\..\..\..\parserComponents\windows\source
```

Then click on OK.

The /I means to use the directory as an include directory. The four sets of double dots are necessary because, apparently, the compilation is attempted from the <NDTS>\tutorials\windows\analyze\analyze directory.

11. To make the executable analyze, select Build Solution from the Build menu. The executable will appear in <NDTS>\tutorials\windows\analyze\Debug\analyze.exe.

Windows will build executables in either the *debug* mode (meaning to build an executable with debugging code built in) or in the *release* mode (meaning to build an executable without debugging code). In order to run an executable built in the *debug* mode, it is necessary to have Visual C++ installed. If the executable is intended to run on a computer



that does not have Visual C++ installed, *release* mode must be used when compiling. The Windows default mode is *debug*, and the tutorial projects use this setting.

The utilities in the test suite are built in *release* mode, and the mode is set to *release* in the projects that build the utilities. To change from one mode to another, before selecting Build Solution from the Build menu, select Configuration Manager from the Build menu. A popup window will appear. In the Configuration column, select either Debug or Release. Then close the popup.

When an executable is built in *release* mode, it appears in the project's top-level **Release** directory. For example, if it is built in *release* mode, the analyze tutorial will appear in <NDTS>\tutorials\windows\analyze\Release\analyze.exe. A lower level Release directory will also be created elsewhere in the project; don't let that confuse you.

The time required for compiling is much longer in *release* mode – up to ten minutes on a Dell Dimension 8300 running Windows XP for the dmisConformanceChecker.

12. Select Save All from the File menu, then select Exit from the File menu.