System Builders Manual for Version 2.1.5 of the NIST DMIS Test Suite (for DMIS 5.1)

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 7610
August 19, 2009

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 70NANB6H013.

Table of Contents

1 Introduction

1.1 Overview

This is a system builders manual for the NIST DMIS Test Suite, version 2.1.5[1]. The purpose of this manual is to help system builders use software provided in the test suite for building systems that implement version 5.1 of DMIS (the Dimensional Measuring Interface Standard).

The test suite and this manual were prepared at the National Institute of Standards and Technology (NIST). There is also a “Users Manual for Version 2.1.5 of the NIST DMIS Test Suite (for DMIS 5.1)”. 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 “prismatic2 conformance class” and “parser” that are explained in the users manual. Also, the users manual has information about compiling the libraries and parsers. The test suite, which includes both manuals, may be downloaded from

http://www.isd.mel.nist.gov/projects/metrology_interoperability/dmis_test_suite.htm

In addition, since the test suite is over 100 megabytes (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 (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 4 (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 5 (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.

1.3 Compiling the Tutorials

The tutorial programs may be compiled using any modern C++ compiler. The tutorials are already

1. In the remainder of this manual “the test suite” means the NIST DMIS Test Suite, version 2.1.5.

compiled, 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.

1.3.1 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

1.3.2 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

1.3.3 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.

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

There are four sets of C++ classes that represent DMIS in the parserComponents directory of the test suite, one for full DMIS and one for each of the three prismatic conformance classes. You, the system builder, should use the set for the conformance class you want to implement[1]. If you are

1. You can use the C++ classes for full DMIS for any conformance class. If you do that and you are parsing, however, you will have to add your own code for conformance class checking.

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.

Each set of C++ classes is described in two files: a header file defining the classes and a code file that implements the functions and methods declared in the header file. For full DMIS these are named dmisFull.hh and dmisFull.cc. For the prismatic2 conformance class, they are named dmisPrismatic2.hh and dmisPrismatic2.cc. The names are similar for the other two prismatic conformance classes. In Windows, the suffixes are changed to .h and .cpp.

In the remainder of this section, the classes for full DMIS will be used. Wherever “full” appears in a name, just remember that it could be replaced by “prismatic1”, “prismatic2”, or “prismatic3”.

Also:

- DMIS code is shown in `this font`.
- C++ code is shown in this font.
- DEBNF code is shown in *`this font`*.

Section 6.5 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[1].
- 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 each DMIS conformance class were built automatically from the DEBNF for the conformance class. All the C++ names of classes and attributes are derived from names used in the DEBNF. Section 2.3 gives details on how attributes are named. Section 2.4 gives details of how classes 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

There is a single class, dmisFullCppBase, at the root of the class hierarchy. All other classes are derived directly or indirectly from dmisFullCppBase. It exists in order to introduce the virtual function printSelf. The dmisFullCppBase class is shown in Figure 1.

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

Figure 1. dmisFullCppBase Class

When a class is constructed from a production that has more than one definition, it is a parent class and has the form shown in Figure 2. The prototype of a parent class is shown on the left side of the figure, and an example of the prototype, boundMinor, is shown on the right.

```
class aClass :                                    class boundMinor :
  public parentClass                                public dmisFullCppBase
{                                                 {
public:                                           public:
  aClass();  <------- constructor with no arguments ------> boundMinor();
  ~aClass();  <------------ destructor ------------> ~boundMinor();
  void printSelf() = 0;  <-- virtual printSelf method --> void printSelf() = 0;
};                                                };
    PROTOTYPE                                         EXAMPLE
```

Figure 2. C++ Parent Class

When a class is constructed from a production that has a single definition, the class has the form shown in Figure 3. A class whose structure follows this prototype, callMacro, is shown in Figure 5. callMacro has three attributes.

```
class aClass :
  public parentClass
{
public:
  aClass();                                          // constructor with no arguments
  aClass(                                            // constructor with arguments
    firstType * a_firstTypeIn,                         // first argument to constructor
    …
    lastType * a_lastTypeIn);                        // last argument to constructor
  ~aClass();                                         // destructor
  void printSelf();                                  // printSelf method
  firstType * get_a_firstType();                     // get function for first attribute
  void set_a_firstType(firstType * a_firstTypeIn);   // set function for first attribute
  …
  lastType * get_a_lastType();                       // get function for last attribute
  void set_a_lastType(lastType * a_lastTypeIn);      // set function for last attribute
private:
  firstType * a_firstType;                           // first attribute
  …
  lastType * a_lastType;                             // last attribute
};
```

Figure 3. Prototype C++ Class

The printSelf function declared in Figure 3 is implemented for all such classes in dmisFull.cc (for Windows, dmisFull.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 sets nothing.

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 nothing.

The prototype in Figure 3 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,

- a public set_attribute function,
- an argument to the constructor that takes arguments.

If 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.

Except when an attribute of a class is a bool or a non-optional int or double, the value of every attribute is a pointer of some sort. Where 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 command names and commas that are always the same in every instance of a class are not represented as attributes of the class. Figure 4 shows the definition of the boundStm class as given in the dmisFull.hh. As shown in Figure 4, 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.

```
/* boundStm

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

BOUND '/' boundMinor #

*/

class boundStm :
  public dmisStatement
{
public:
  boundStm();
  boundStm(
    boundMinor * a_boundMinorIn);
  ~boundStm();
  void printSelf();
  boundMinor * get_a_boundMinor();
  void set_a_boundMinor(boundMinor * a_boundMinorIn);
private:
  boundMinor * a_boundMinor;
};
```

Figure 4. boundStm C++ Class

If part of a DMIS command is optional and contains items that can differ between instances of the command, then there is an attribute (which is a pointer) for each of those items. If an instance of

the command does not include the optional part, then in the class instance representing the command, the pointers for the items in the optional part are NULL pointers.

If part of a DMIS command is optional but contains only items such as keywords and commas that do not differ between instances of the command that have the optional part, then there is a boolean attribute for the optional part.

The callMacro class shown in Figure 5 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` command 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 command will be a pointer to the string "3, 2.5". If not the value will be NULL. If the command 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 * a_mLabelIn,
    char * a_stringIn);
  ~callMacro();
  void printSelf();
  bool get_has_EXTERN();
  void set_has_EXTERN(bool has_EXTERNIn);
  mLabel * get_a_mLabel();
  void set_a_mLabel(mLabel * a_mLabelIn);
  char * get_a_string();
  void set_a_string(char * a_stringIn);
private:
  bool has_EXTERN;
  mLabel * a_mLabel;
  char * a_string;
};
```

Figure 5. callMacro C++ Class

2.3 Attribute names

The names of most attributes are formed by concatenating the prefix a_ with the type of the data. For example, in Figure 5, the attribute name a_mLabel is used where the data type is mLabel. If there are two or more occurrences of the same type of data in a class, the prefix is not used. A suffix of the form _N is added instead, where N is an integer giving the position of the attribute in the DEBNF being represented. For example, *daLabel ',' daLabel* is the DEBNF for *equateMinor_1*. The attributes of equateMinor_1 are named daLabel_1 and daLabel_3 since they are the 1st and 3rd items in the DEBNF.

If the data type is char *, then 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 or the _N suffix 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<dmisItem *> has the name a_dmisItemList. 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.

If the data type is bool, then the attribute name is made by concatenating the prefix has_ with the first name in the optional items being represented. For example, in Figure 5, the attribute that indicates whether `EXTERN,DMIS,` is used is named has_EXTERN. If there are two or more attributes that would have the same name, then the _N suffix is added as described above (and the has_ prefix is kept).

2.4 Class names

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

If 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 if 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.

For example, if the DEBNF is
pameasRotaryAngle = rotAbs , c , angle | rotIncr , c , angle ;
then the class names will be pameasRotaryAngle_rotAbs and pameasRotaryAngle_rotIncr.

(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
aclratMeas = MESACL , c , aclratLinear | MESACL , c , aclratDef ;
then the class names will be aclratMeas_aclratLinear and aclratMeas_aclratDef.

If none of the three conditions above holds, then the class name has the form productionName_N, where N is 1 for the first definition, 2 for the second definition, etc.

For example, if the DEBNF is
labelName = labelNameConst | '(' , '@' , stringVar , ')' ;
then the class names will be labelName_1 and labelName_2.

2.5 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 command, however, two documents are needed: (1) a copy of the DMIS 5.1 standard (electronic or paper) and (2) an electronic copy of the header file defining the classes. Since there are a lot of classes (1838 for full DMIS, covering almost 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 command is to begin by searching the header file for class *command*Stm, where *command* is the name of the command in lower case letters. For example, if you want to find the classes for the BOUND command, search for class boundStm. That will get you very quickly to the class for the command. In a few cases, that is all you need to do. In most cases, however, the class for the command 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 the BOUND command `BOUND/F(f1),F(f2),F(f3)`, for example, a tree of class instances is shown in Figure 6. In the figure, attribute names and list indexes are in *italics*. The tree includes three instances of fLabel.

boundStm
    a_boundMinor boundFeat
        a_fLabel fLabel
            a_labelName labelName_1
                a_labelNameConst labelNameConst
                    a_string “f1”
        a_featureList std::list<featureLabel *>
            1 fLabel
                a_labelName labelName_1
                    a_labelNameConst labelNameConst
                        a_string “f2”
            2 fLabel
                a_labelName labelName_1
                    a_labelNameConst labelNameConst
                        a_string “f3”

Figure 6. Instance Tree for BOUND/F(f1),F(f2),F(f3)

As shown in Figure 6, instances of six classes are used in constructing this bound statement: boundStm, boundFeat, fLabel, labelName_1, labelNameConst, and featureLabel (in std::list<featureLabel *>). The C++ code (from dmisFull.hh) for the first two of these (and boundMinor) is shown in Figure 7.

```
/* boundStm - represents: BOUND '/' boundMinor #*/
class boundStm :
  public dmisFreeStatement,
  public dmisStatement
{public:
  boundStm();
  boundStm(
    boundMinor * a_boundMinorIn);
  ~boundStm();
  void printSelf();
  boundMinor * get_a_boundMinor();
  void set_a_boundMinor(boundMinor * a_boundMinorIn);
private:
  boundMinor * a_boundMinor; };

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

/* boundFeat - represents: fLabel ',' featureList*/
class boundFeat :
  public boundMinor
{public:
  boundFeat();
  boundFeat(
    fLabel * a_fLabelIn,
    std::list<featureLabel *> * a_featureListIn);
  ~boundFeat();
  void printSelf();
  fLabel * get_a_fLabel();
  void set_a_fLabel(fLabel * a_fLabelIn);
  std::list<featureLabel *> * get_a_featureList();
  void set_a_featureList(std::list<featureLabel *> * a_featureListIn);
private:
  fLabel * a_fLabel;
  std::list<featureLabel *> * a_featureList;};
```

Figure 7. C++ Some Classes for BOUND/F(f1),F(f2),F(f3)

The boundStm instance has one attribute, *a_boundMinor*, and its value is an instance of 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, labelName_1, and labelNameConst is left to the reader.

2.6 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 one of the C++ classes for DMIS is polymorphic, so dynamic_cast will always work on them. 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:

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

You can, of course, combine the assignment and the test, in which case you do not use isA at all. As shown below, this is shorter but a little obscure.

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

2.7 Parse Tree

When the yyparse 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 inputFile. 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 following sort:

```
void doItems(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))
      …
  }
}
```

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

The “analyze” tutorial described in Section 5 uses an iterator and a for loop of the sort shown

above.

3 The “makeBound” Tutorial Program

3.1 What the Program Does

The makeBound tutorial program shown in Figure 8 generates and prints the instance of the boundStm class shown in Figure 6. The code may also be found in tutorials/linuxSun/source/ makeBound.cc and tutorials/windows/source/makeBound.cpp. This tutorial is written for the full DMIS conformance class.

```
#include "dmisFull.hh"

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

      featureLabels = new std::list<featureLabel *>;
      featureLabels->push_back(new fLabel
            (new labelName_1
                  (new labelNameConst(featLabel1))));
      featureLabels->push_back(new fLabel
            (new labelName_1
                  (new labelNameConst(featLabel2))));
      return new boundStm
            (new boundFeat
                  (new fLabel
                        (new labelName_1
                              (new labelNameConst(boundLabel))),
                  featureLabels));
}

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

Figure 8. C++ Program to Make BOUND/F(f1),F(f2),F(f3)

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 with the same hierarchy as shown in Figure 6. There is no constructor for a populated list, however, so the list is built and populated (using push_back) before the boundStm is built.

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

3.2 How to Run the Program

3.2.1 Linux

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

binLinux/makeBound

3.2.2 Sun

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

binSun/makeBound

3.2.3 Windows

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

Debug\makeBound

4 The "Generate" Tutorial Program

4.1 What the Program Does

The "generate" tutorial program builds a specific DMIS input file (the one in systemTestFiles/prismatic2/okInMotionP2/simple1p2.dmi, but without comments). It is written for the prismatic2 conformance class. The source code for the program is in tutorials/linuxSun/source/generate.cc and in tutorials\windows\source\generate.cpp. The "generate" tutorial 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 8. 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 labelName_1 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 dmisPrismatic2.hh and dmisPrismatic2.a are used in compiling the program. For Windows, the files dmisPrismatic2.h and dmisPrismatic2.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/generate

4.2.2 Sun

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

binSun/generate

4.2.3 Windows

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

Debug\generate

5 The “Analyze” Tutorial Program

5.1 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 kind of nominal feature 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 dmisPrismatic2YACC.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 yyparse; 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 dmisItemList 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.

This tutorial program uses files for the prismatic2 conformance class, but any other conformance class (including full DMIS) could be used equally well. The features the program looks for would need to be changed to be the ones in the conformance class. For Linux and Sun, the files dmisPrismatic2.hh and dmisPrismatic2.a are used in compiling the program. For windows, the files dmisPrismatic2.h and dmisPrismatic2.lib are used.

5.2 How to Run the Program

The “runAllPrismatic2” file contains a list of the names of DMIS input files, so “runAllPrismatic2” 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 files names are complete relative or absolute path names.

5.2.1 Linux

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

binLinux/analyze runAllPrismatic2

5.2.2 Sun

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

binSun/analyze runAllPrismatic2

5.2.3 Windows

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

Debug\analyze runAllPrismatic2

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 1.3.3. 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”
- “full” for “prismatic2”
- “dmisFull” for “dmisPrismatic2”

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\NistDmisTestSuite2.1.5

   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>\parserComponents\windows\prismatic2\source directory, select the following source code files, and then press Add:

dmisPrismatic2.h

From the <NDTS>\tutorials\windows\source directory, select the following source code files, 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 dmisPrismatic2 library, in the Project menu of the main window, select Add Existing Item. This brings up a file browser window.

   From the <NDTS>\parserComponents\windows\prismatic2\dmisPrismatic2Classes\Debug directory, select dmisPrismatic2.lib and then press Add.

   When you add dmisPrismatic2.lib, Visual C++ will display a popup window asking if you want to create a rule for making dmisPrismatic2.lib.

   Press the No button.

   In the Solution Explorer window, dmisPrismatic2.lib goes directly into dmisFullParser, not in any fake directory.

10. Even through Visual C++ knows exactly where the dmisPrismatic2.h file is (and will display it if you double click on it in the Solution Explorer window), Visual C++ does not find dmisPrismatic2.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\prismatic2\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 the file <NDTS>\tutorials\windows\analyze\Debug\analyze.exe.

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