Users 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 7735
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 Overview

This manual is a users manual for the NIST DMIS Test Suite, version 2.2.1. The test suite[1] is intended to serve two purposes:

- to help users and vendors use version 5.2 of DMIS (the Dimensional Measuring Interface Standard)
- to provide utilities and test files for conducting conformance tests on
  - DMIS input files
  - computer systems that generate DMIS input files
  - computer systems that execute DMIS input files.

The test suite and this manual were prepared at the National Institute of Standards and Technology (NIST). There are also a "System Builders Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2)"[2] and a "Maintainers Manual for the NIST DMIS Test Suite, Version 2.2.1". The purpose of the system builders manual is to help system builders use software provided in the test suite for building systems that implement DMIS. 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

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.

DMIS is the only international standard language (ISO 22093) for input files (programs) used for the control of dimensional measuring equipment, coordinate measuring machines in particular. It is also approved by the American National Standards Institute (ANSI/DMIS 105.2, Part 1-2009). The most recent version of DMIS approved by the Dimensional Metrology Standards Consortium (DMSC), the organization accredited by ANSI that built and maintains DMIS, is DMIS 5.2. Copies of the ANSI standard on CD are available for purchase at http://www.dmisstandards.org/store. Copies of the ISO standard version are expected to be available soon from ISO.

Because DMIS is a very large language, and only subsets of it need to be implemented for many applications, subsets called conformance classes have been defined. To conform to a conformance class, a system using DMIS must fully implement the subset for that class. The DMSC has defined two "application profiles", one for Prismatic parts and one for Thin Walled parts. Seven addenda have also been defined. Each application profile (AP) and addendum may be implemented at three levels. Level 2 includes everything in level 1, plus additional items. Level 3 includes everything in level 2, plus additional items. This version of the test suite can handle full DMIS or any combination of an AP and 0 to 7 addenda at any of the three levels. A "conformance module" is a level of an AP or an addendum. With two APs plus seven addenda at three levels, there are 27 conformance modules. A conformance class may be defined by specifying conformance modules.

1. In the remainder of this manual "the test suite" means the NIST DMIS Test Suite, version 2.2.1.
2. In the remainder of this manual "the system builders manual" means the System Builders Manual for Version 2.2.1 of the NIST DMIS Test Suite (for DMIS 5.2).

EBNF (Extended Backus-Naur Form) is a standard formal language for defining the syntax of a language (ISO/IEC 14977). Annex C of the DMIS standard is an EBNF definition of the syntax of the DMIS input language. The term DEBNF (short for DMIS EBNF) is used in this manual to mean the dialect of EBNF used in DMIS 5.2. Details are given in Section 10.

As shown in Figure 1, the test suite has eight directories.

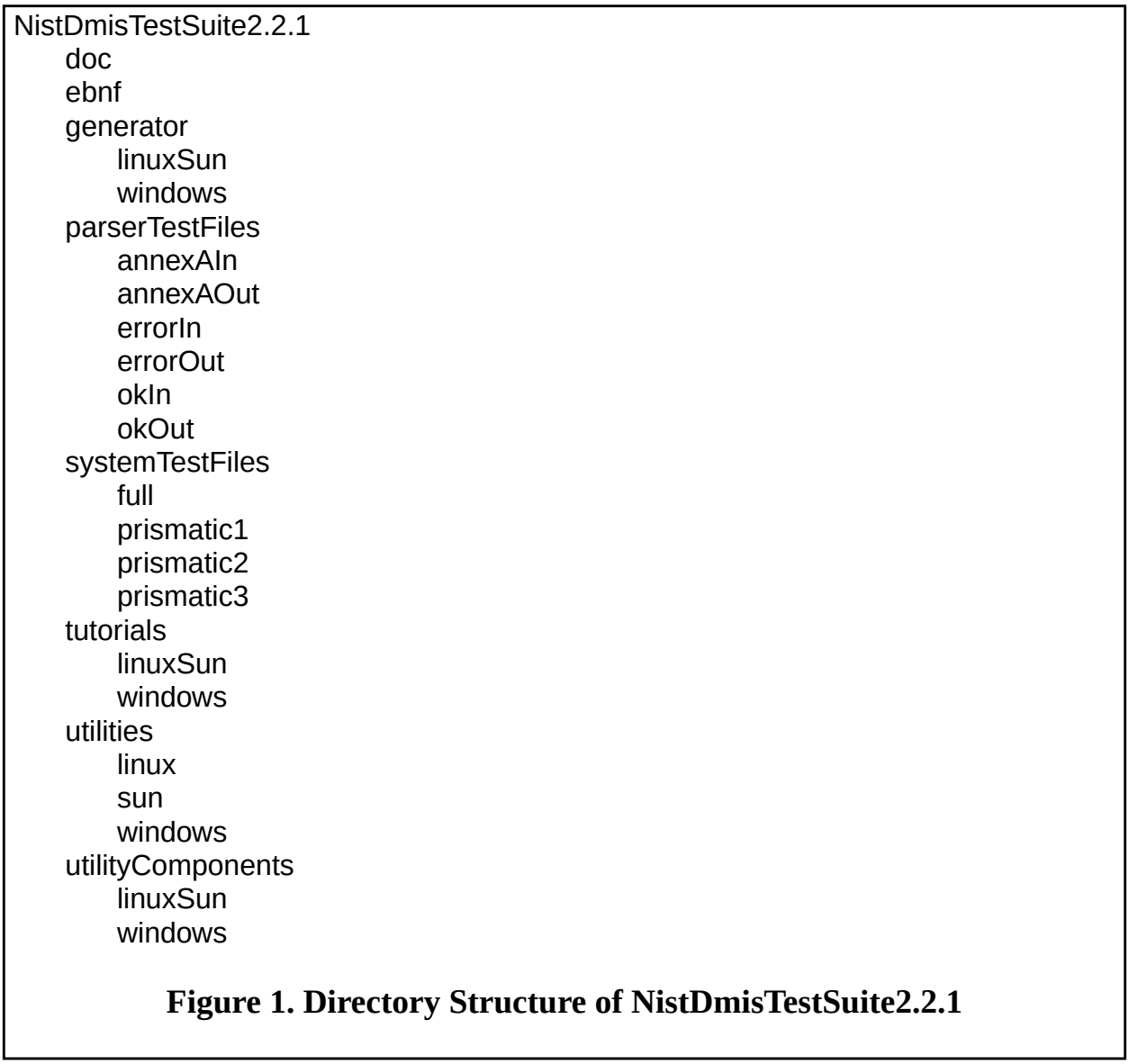

Figure 1. Directory Structure of NistDmisTestSuite2.2.1

Briefly, these contain the following.

The doc directory has this users manual, the system builders manual, the test suite maintainers manual, and an Excel spreadsheet defining conformance classes.

The ebnf directory includes the DEBNF file for full DMIS.

The generator directory contains a system named debnf2pars for automatically generating much

of the code in the utilityComponents directory. The input to debnf2pars is a DEBNF file. The generator directory also contains a system named generateMore for automatically generating more of the code in the utilityComponents directory. The generator directory is not expected to be of interest to most users. See the Maintainers Manual for details.

The utilityComponents directory has software from which a library and five utilities can be built in the utilities directory for each of three operating systems.

The utilities directory contains the five executable utilities for each of three operating systems (Linux, SunOS, and Windows). For running tests on the utilities, the utilities directory also contains scripts, C++ code, executables built from the C++ code, and data files. The utilities are made from the code in the utilityComponents directory. The utilities are:

- dmisParser - checks that a single DMIS input file or each of a set of files conforms to the syntax of full DMIS. The dmisParser prints the number of errors and warnings found in each file, and pretty-prints the file. Details about the dmisParser are given in Section 3.
- dmisConformanceChecker - runs two ways. If given a single DMIS input file, it checks the file against the syntax requirements of full DMIS or any allowed combination of the 27 conformance modules. If given a file containing the names of a number of DMIS input files, it does the same kind of check for each input file and then prints the number of times each DMIS statement was used. Details about the dmisConformanceChecker are given in Section 4.
- dmisConformanceRecorder - reads and checks a single DMIS input file, prints the number of errors and warnings found, and inserts a conformance statement on the `DMISMN` or `DMISMD` line of the file. The conformance statement specifies which conformance modules are needed to handle the file. Details about the dmisConformanceRecorder are given in Section 5.
- dmisConformanceTester - reads and checks a single DMIS input file the way the dmisConformanceChecker does, then behaves like the dmisConformanceRecorder except that it prints the conformance statement only in the command window (not in the file) and it identifies what DMIS construct first caused each conformance module to be required. Details about the dmisConformanceTester are given in Section 6.
- dmisTestFileReductor - reads one or more specially marked parser test files for full DMIS and produces corresponding parser test files for a conformance class of DMIS (i.e., a subset of DMIS). Details about the dmisTestFileReductor are given in Section 7.

The first four of these utilities use an underlying parser that reads a DMIS input file, reports syntax errors and warnings, and builds a parse tree if there are no errors.

The parserTestFiles directory has a lot of DMIS input files for testing utilities. These are syntactically correct but do not necessarily make sense as programs.

The systemTestFiles directory has a modest number of DMIS input files that should be executable on commercial DMIS execution systems.

The tutorials directory, which is described in the System Builders manual, contains:

- one tutorial (makeBound) showing how to use the C++ classes for DMIS (given in the utilityComponents directory) for building a single line of DMIS code,
- one tutorial (generate) showing how to use the C++ classes for DMIS (given in the utilityComponents directory) for building a DMIS input file generator, and
- one tutorial (analyze) showing how to use the parser and the C++ classes for DMIS (given in the utilityComponents directory) for building a DMIS input file consumer.

1.2 Downloading and Installing the Test Suite

The test suite may be downloaded from:

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

Select and download the file NistDmisTestSuite2.2.1.zip. Then unzip the file.

The top level directory that will be created is named NistDmisTestSuite2.2.1. The structure of this directory is shown in Figure 1 above. It should not be necessary to recompile any of the executables in the test suite. If you do want to recompile, see the instructions in Section 1.4 of the System Builders Manual.

1.3 Documentation

The doc directory contains a copy of this users manual, a copy of the system builders manual, a copy of the maintainers manual, and an Excel spreadsheet defining the DMIS conformance modules for DMIS 5.2. The doc directory does not contain a copy of the ISO standard for EBNF (ISO/IEC 14977), but that may be downloaded free of charge from ISO at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

The Excel spreadsheet is important because it is the controlling document for the definition of the conformance modules. All of the C++ code defining conformance modules was prepared by first hand-editing C++ files so that they say the same thing as the spreadsheet.

1.3.1 How is the Excel spreadsheet defining conformance classes arranged?

DMIS51ProfileTemplates_MAR2008_Draft2.xls

1		L1	Essential to meeting the Profile's goals	Prismatic Application Profile				Thin Walled Application Profile				Rotary Table AP Addendum					
2		L2	Important to meeting the Profile's goals														
3		L3	Beneficial to meeting the Profile's goals.														
4		N/A	Not applicable to this Profile														
5																	
6	COMMAND	PARAMETERS		L1	L2	L3	N/A	L1	L2	L3	N/A	L1	L2	L3			
7	ACLRAT	ACLRAT/var_1				x				x				x			
8		var_1	MESACL,var_2			x				x							
9			POSACL,var_2			x				x							
10			ROTACL,var_3				a				a			x			
11		var_2	MPMM,n			x				x							
12			MMPSS,n			x				x							
13			IPMM,n			x				x							
14			IPSS,n			x				x							
15			var_4			x				x							
16		var_3	RPMM,n				a				a			x			
17			var_4				a				a			x			
18		var_4	PCENT,n			x				x				x			
19			HIGH			x				x				x			
20			LOW			x				x				x			
21			DEFAULT			x				x				x			
22	ALGDEF	VA(label)=ALGDEF/var_1					x				x						
23		var_1	CODE,n				x				x						
24			'name' var_2				x				x						
25		var_2	,parameter var_2				x				x						
26			does not exist				x				x						
27	ASSIGN	varname=ASSIGN/expr			x	x			x	x							
28	BADTST	BADTST/var_1			x	x			x	x							
29		var_1	ON		x	x			x	x							
30			OFF		x	x			x	x							
31	BOUND	BOUND/var_1 var_2 var_3		x	x	x			x	x							
32		var_1	F(label1)	x	x	x			x	x							

Figure 2. Excel SpreadSheet

The arrangement of the Excel spreadsheet defining conformance classes is largely self-evident, so it is only summarized here. Figure 2 shows the left side of the beginning of the spreadsheet.

The spreadsheet has a main row for each subsection of section 6 of DMIS 5.2. Each row is divided into subrows that follow the structure of the syntax described in the corresponding subsection of DMIS 5.2. Main rows for ACLRAT, ALGDEF, ASSIGN, BADTST, and BOUND are shown on Figure 2.

The spreadsheet has 11 main columns, the first five of which are shown on Figure 2. The first main column has the DMIS statement names. For each statement, the second column has a list of the parameters used by the statement, one subrow for each parameter. The rest of the columns also have these subrows.

Columns 3 and 4 are for the Prismatic Application Profile and the Thin Walled Application Profile. These two columns are each divided into four subcolumns. The first three subcolumns are for the three levels at which the two application profiles may be implemented. An **x** in a subrow and subcolumn means the parameter in the subrow must be implemented at the level of the subcolumn. The meaning of the fourth subcolumn is a little trickier. An **x** in the fourth subcolumn means the parameter does not have to be implemented in any application profile or addendum. An **a** in the fourth subcolumn means that the parameter must be implemented in at least one addendum (and in this case an **x** will be found in the same subrow for one or more of the addenda).

The last seven columns of the spreadsheet are for the addenda, and each one is divided into three subcolumns for the levels. An **x** in any of these columns means the parameter must be implemented. The addenda are:

- Rotary Table (RY) - the only one shown on Figure 2
- Multi Carriage (MC)
- Contact Scanning (CT)
- In-Process Verification (IP)
- Quality Information System (QI)
- Measurement Uncertainty (MU)
- Soft Gaging (SF).

1.4 Arrangement of this Manual

Sections 2 through 7 of this manual cover the utilities, how to use them, and how they work. Section 8 (“Parser Test Files”), and Section 9 (“System Test Files”) describe the test files included in the test suite. Section 10 (“EBNF”) describes EBNF, the formal language used to define the syntax of DMIS unambiguously. The C++ classes for DMIS and the tutorials are covered in the system builders manual. The software of the test suite is covered in the maintainers manual.

1.5 Use of Fonts

To help make it clear what sort of thing is being discussed, in this manual:

- DMIS code and messages printed by the utilities in the test suite are shown in `this font`.
- File and directory names (including names of executable files) and internet addresses are shown in this font.
- EBNF code is shown in **`this font`**.
- Commands typed in a command window are shown in ***this font***.

1.6 Changes from Version 2.1.5

Major changes have been made from the NIST DMIS Test Suite Version 2.1.5 (the most recent earlier release placed on the web) as follows. Be aware also that Version 2.1.5 made major changes from Version 2.1.4, most of which have been carried over into Version 2.2.1.

1.6.1 DMIS 5.2

Version 2.2.1 deals with DMIS 5.2. Version 2.1.5 dealt with DMIS 5.1. Everything in Version 2.2.1 that needed to be updated has been updated.

1.6.2 Full Unified Coverage

In Version 2.1.5, only four conformance classes were covered (full DMIS plus prismatic 1, 2, and 3), and there were separate parsers, DEBNF files, and C++ code files for each of the four. In Version 2.2.1, there is only one DEBNF file, and all 98,305 conformance classes are covered. Also, there is only one executable for each utility. There is a C++ code file for each of the 27 conformance modules, however.

1.6.3 More Utilities

Version 2.2.1 has five different utilities where Version 2.1.5 had only one kind of utility (parsers). In Version 2.2.1, the functionality of counting DMIS statements has been moved into the dmisConformanceChecker. That functionality was included in the parsers in Version 2.1.5.

1.6.4 DEBNF

1.6.4.1 more DMIS statements

In Version 2.1.5, there was one DMIS statement (featStm) for all feature types and one (tolStm) for all tolerance types. In Version 2.2.1, there are 28 feature statements and 29 tolerance statements. Since reporting in the dmisConformanceChecker is done at the DMIS statement level, this provides for much more detailed reporting from that utility.

1.6.4.2 condensation

In a number of cases, the DEBNF for a DMIS statement has been changed by condensing two DEBNF productions into one. This leads to a more compact and efficient C++ class hierarchy.

1.6.4.3 naming

A method of specifying meaningful names has been implemented. The names are introduced in the DEBNF by giving them as DEBNF comments. The names are propagated from the DEBNF into C++ code by the debnf2pars generator. For example, in Version 2.2.1 the attributes of a C++ doStm are index, init, limit, and incr, whereas in Version 2.1.5 they were a_intVar, intVal_3, intVal_5 and intVal_7.

1.6.5 Generator

1.6.5.1 debnf2pars

The debnf2pars generator in Version 2.2.1 generates everything that was generated in Version 2.1.5 (except that printing a page or two of fixed code has been removed). In addition, Version 2.2.1 generates C++ code files for three of the new utilities.

1.6.5.2 generateMore

In Version 2.2.1, manual editing of the 27 conformance module C++ files is required after files are generated by debnf2pars. Further automatic processing of those files is done after the editing by the generateMore utility, which produces one more C++ file (assignModuleSubAtts.cc).

1.6.6 Parser Test Files

Version 2.2.1 has parser test files only for full DMIS, whereas Version 2.1.5 had test files for full DMIS plus the three levels of the prismatic AP. As described in Section 7, however, the dmisTestFileReductor produces parser test files automatically for any allowed set of conformance modules. To enable this, every line of every one of the 254 test files in the parserTestFiles/okIn directory has been marked with a DMIS comment indicating the conformance modules needed to deal with the line.

1.6.7 Testing

1.6.7.1 cycle testing of conformance information

To check that the dmisConformanceRecorder works properly, a cycle test was devised in which each of over 300 test files had its conformance information inserted without using the dmisConformanceRecorder. Then each file was run through the dmisConformanceRecorder, which replaced the conformance information, and it was checked that the file was unchanged.

1.6.7.2 additional test scripts

Test scripts were written that test the new utilities.

1.7 Shortcomings

The largest shortcomings of the test suite are:

- The current test suite tests only the compliance of DMIS input files to the requirements for input file syntax. There is no semantic checking whether the input files describe useful, logical, or realizable measurement operations.
- There is currently no test utility for verifying the compliance of a DMIS execution system to the requirements for system behavior.
- For testing completeness of a DMIS generator, the dmisConformanceChecker counts only the number of times each DMIS statement is used. For a comprehensive completeness test, it would be necessary to implement counting the number of times every C++ class and attribute in a conformance class is used. The underlying software of Version 2.2.1 will support that test, but the debnf2pars generator has not been modified so that it will generate code to implement that test.
- The utilities do not attempt to load files referenced by `INCLUD` or `EXTFIL`, so the utilities may report errors that would not occur if such files were used as provided by those DMIS statements.
- The utilities check all file lines sequentially, so if a file uses flow of control statements (such as `IF/ELSE/ENDIF` or `JUMPTO`) that cause a file to be executed out of order or cause some statements not to be executed, the utilities may report warnings incorrectly or may fail to detect errors for which warnings should be given.
- The utilities do not evaluate variables, so labels that are created or referenced using the @variable method may cause the parser to report label warnings incorrectly.

2 Utilities

2.1 What is in the utilities directory?

The utilities directory contains executable utilities. For running tests on the utilities, the utilities directory also contains scripts, C++ code, executables built from the C++ code, and data files.

2.2 How is the utilities directory arranged?

The structure of the utilities directory is shown in Figure 3 below. There is a subdirectory for each of three operating systems: Linux, SunOS, and Windows XP. Each of these has bin, code, and full subdirectories.

Each bin subdirectory contains executables for the five utilities (dmisParser, dmisConformanceChecker, dmisConformanceRecorder, and dmisConformanceTester) plus other executables used in the test scripts.

Each code subdirectory contains C++ source code for the executables used in the test scripts. For Windows, the code subdirectory also contains a subdirectory for each of the five executables used in the test scripts. These were built by Visual C++ during the process of compiling the executables.

Each full subdirectory contains

- a checkLevels script that runs each of 306 DMIS input files through the dmisConformanceRecorder and verifies that the file did not change even though the conformance information was replaced
- a testFullParser script that runs 322 DMIS input files through the dmisParser and verifies that for each of those files, the parser output is what it is expected to be
- a testFullTester script that runs 306 DMIS input files through the dmisConformanceTester (but does not test that the conformance information produced by the dmisConformanceTester is correct)
- a runAllFull file that contains the names of 322 DMIS input files and is intended to be used for testing the dmisParser and the dmisConformanceChecker
- a runAllFullOut file containing what the dmisConformanceChecker should print when it processes the runAllFull file using the command ***dmisConformanceChecker runAllFull***,
- a runAllFullOut3 file containing what the dmisConformanceChecker should print when it processes the runAllFull file using the command ***dmisConformanceChecker runAllFull PM3 RY3 CT3 MC3 IP3 QI3 MU3 SF3***,
- a runOkFull file that contains the names of 254 DMIS input files and is intended to be used with the dmisTestFileReductor as described in Section 7
- a runSomeFull file that contains the names of 143 DMIS input files and is used in the dmisConformanceChecker examples in Section 4.2
- an empty outgoing directory.

The full subdirectory in the utilities/windows directory includes four more .bat files used by the primary scripts.

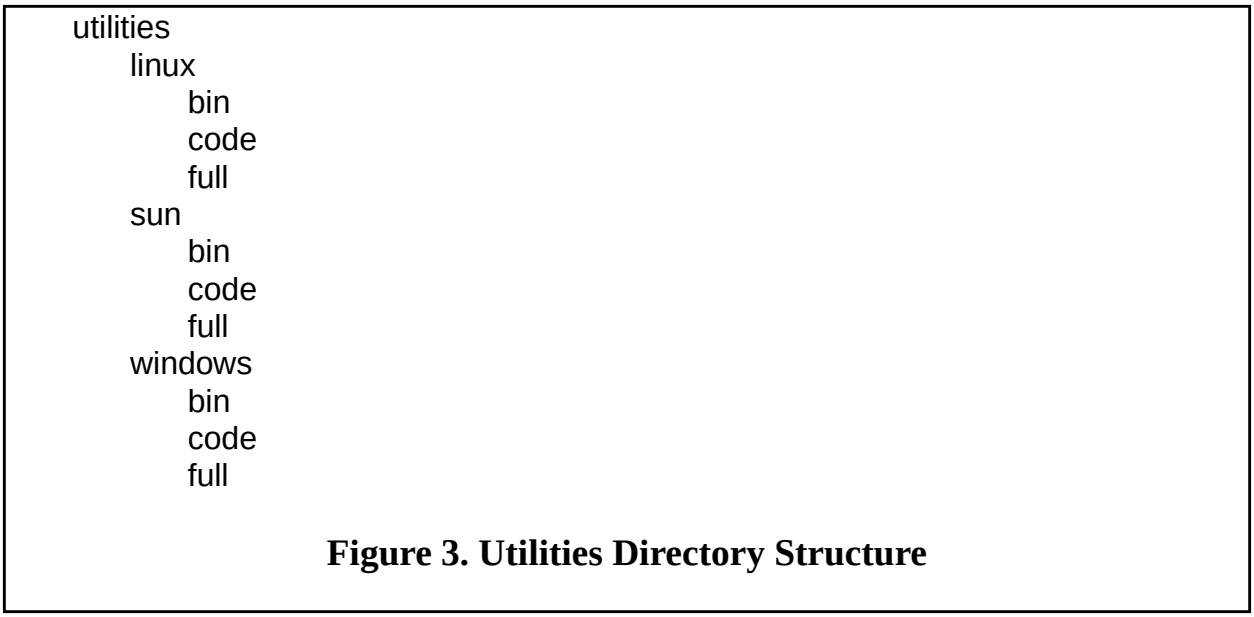

Figure 3. Utilities Directory Structure

2.3 Which utility should I use?

If you want to check that DMIS input files conform to full DMIS, use the dmisParser.

If you want to check that DMIS input files conform to a conformance class of DMIS, or if you want to gather information on the use of DMIS statements in a set of DMIS input files, use the dmisConformanceChecker.

If you want to have conformance information inserted automatically on the `DMISMN` or `DMISMD` line of a DMIS input file, use the dmisConformanceRecorder. The conformance information identifies which conformance modules are needed to deal with the file.

If you want to know what the dmisConformanceRecorder will do but you don't want your DMIS input file changed, use the dmisConformanceTester. The dmisConformanceTester also shows why each conformance module is required.

If you want to generate a set of test files for a conformance class defined by a set of DMIS conformance modules, use the dmisTestFileReductor as described in Section 7.

3 The dmisParser

The dmisParser checks that DMIS input files conform to full DMIS.

3.1 How do I use the dmisParser to parse a single DMIS input file?

For all three operating systems, you run the dmisParser by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

The dmisParser takes one command argument, which is either (1) the path to a DMIS input file to parse or (2) the path to a file containing the paths to a number of DMIS input files.

The dmisParser writes its messages in the command window, but output redirection may be used to send the messages to a file.

3.1.1 Linux and SunOS

Example 1. The following command should be given in the utilities/linux directory if you are using Linux or in the utilities/sun directory if you are using SunOS. The command parses the DMIS input file units1.dmi, writes the messages "`0 errors`" and "`0 warnings`" in the command window, and reprints the text of the input file in the command window. The reprinting is done using the parse tree built during parsing. Comments are not reprinted.

bin/dmisParser ../../parserTestFiles/okIn/units1.dmi

Example 2. The following command should be given in the utilities/linux directory if you are using Linux or in the utilities/sun directory if you are using SunOS. The command parses the DMIS input file units1.dmi, writes the messages "`0 errors`" and "`0 warnings`" in the file u1.out, and also reprints the text of the input file in u1.out. Comments are not reprinted.

bin/dmisParser ../../parserTestFiles/okIn/units1.dmi > u1.out

3.1.2 Windows

Example 1. The following command, given in the utilities\windows directory, parses the DMIS input file units1.dmi, writes the messages "`0 errors`" and "`0 warnings`" in the command window, and reprints the text of the input file in the command window. The reprinting is done using the parse tree built during parsing. Comments are not reprinted.

bin\dmisParser.exe ..\..\parserTestFiles\okIn\units1.dmi

Example 2. The following command, given in the utilities\windows directory, parses the DMIS input file units1.dmi, writes the messages "`0 errors`" and "`0 warnings`" in the file u1.out, and also reprints the text of the input file in u1.out. Comments are not reprinted.

bin\dmisParser.exe ..\..\parserTestFiles\okIn\units1.dmi > u1.out

3.2 How can I use a single command to parse a whole set of DMIS input files?

There are two ways to run a set of DMIS input files through the dmisParser, as described in Section 3.2.1 and Section 3.2.2. In both methods, you type a command in a command window.

3.2.1 First method for running a set of DMIS input files through the dmisParser

In the first method:

- A new dmisParser process is used for each file.
- The message output of the dmisParser is compared with the expected message output.
- If no error messages are given by the dmisParser, the parsed file is printed back out again, the input file is reformatted so that it is in the format used by the dmisParser's printer, and a check is made that the file printed by the dmisParser is identical to the reformatted file.
- The test stops at the first file for which there is a difference between the actual and expected message output or between the printed and reformatted files, if there is any input file for which that happens.

This method is implemented using an executable script file. The name of that file, ***testFullParser***, implies that the dmisParser is being tested. This is correct when you have downloaded the test suite and are testing to see if it runs on your computer. However, once you are satisfied that the dmisParser is working on your computer, you can make a copy of

testFullParser and edit the names of the directories and the files, so that the copy can be used for testing whatever set of DMIS input files you want to test.

3.2.1.1 Linux and SunOS

When using Linux, get into the utilities/linux/full directory. When using SunOS, get into the utilities/sun/full directory.

Give the command:

./testFullParser

The testFullParser script processes 322 DMIS input files. It takes 15 seconds or so to run on a Dell Precision 670 PC running Linux and about 27 seconds on a Sun Fire V215 running SunOS.

3.2.1.2 Windows

In Windows get into the utilities\windows\full directory.

Give the command:

testFullParser.bat

The testFullParser.bat batch file processes 322 DMIS input files. It takes about 110 seconds to run on a Dell Dimension 8300 PC running Windows XP.

3.2.2 Second method for running a set of DMIS input files through the dmisParser

In the second method:

- Only one dmisParser process runs (so the test is faster), and it parses all the files.
- The test stops if a test file cannot be found but does not stop if there is a parse error.
- No comparison is done.

You can make a file listing the names of the DMIS input files you want to test and use this method to test them. The file names must include the path (either a relative path starting from the directory containing the dmisParser or an absolute path).

3.2.2.1 Linux and SunOS

To run a set of DMIS input files through the dmisParser in Linux, get into the utilities/linux/full directory; in SunOS, get into the utilities/sun/full directory.

Example 1. Give the command:

../bin/dmisParser runAllFull

The command runs 322 DMIS input files through the dmisParser. Output printing goes to the command window. The runAllFull file contains the names of the 322 DMIS files. The names of the DMIS files must end in .dmi. On a Dell Precision 670 PC running Linux, this takes about 3 seconds. On a Sun Fire V215 running SunOS, this takes about 6 seconds.

3.2.2.2 Windows

To run a set of DMIS input files through the dmisParser in Windows, get into the utilities\windows\full directory.

Example 1. Give the command:

..\bin\dmisParser.exe runAllFull

The command runs 322 DMIS input files through the dmisParser. Output printing goes to the command window. The runAllFull file contains the names of the 322 DMIS files. The names of the DMIS files must end in .dmi. In Windows, it does not matter whether or not you include the .exe suffix in the name of the command. On a Dell Dimension 8300 PC running Windows XP, this takes about 5 seconds.

3.3 How do I modify a command that runs a set of test files so that it tests my parser?

This is possible only if your parser is an executable that takes arguments and can be run from a command window. If you are using Linux or SunOS you need to know how to write a script file; on Windows you need to know how to write a batch file.

3.3.1 Linux and SunOS

In Linux or SunOS, if you have a DMIS parser, start by copying the testFullParser script file to testMyFullParser (or a name you prefer). Edit testMyFullParser by deleting the "runOut" function defined near the beginning of the file and editing the "runOK" function so that:

- It calls your parser instead of dmisParser.
- It takes the arguments your parser takes.
- It performs the output test(s) you want.

Several other items may be need to be changed, particularly near the beginning of testMyFullParser where variables are defined and in the last section where files with syntactic errors are tested.

3.3.2 Windows

In Windows, if you have a DMIS parser, start by copying the testFullParser.bat batch file to testMyFullParser.bat (or a name you prefer). Edit testMyFullParser.bat by replacing the `TESTOK=call parseOneOK.bat` line near the beginning with `TESTOK=call parseMyOneOK.bat`. Also, delete the `TEST=call parseOne.bat` line. Then write a parseMyOneOK.bat file that:

- calls your parser instead of dmisParser.
- takes the arguments your parser takes.
- performs the output test(s) you want.

Several other items may need to be changed, particularly near the beginning of testMyFullParser.bat where variables are defined and in the last section where files with syntactic errors are tested.

3.4 What does the dmisParser do when it parses a single DMIS input file?

If the first argument to the command to run the dmisParser ends in .dmi, it assumes the file is a DMIS input file and does the following.

When the dmisParser runs, it runs in two stages: preprocess and parse. Wherever there may be an error in the DMIS input file, an error or warning message is printed preceded by the line number (in the input file) of the line that caused the problem. In the parse phase, the text of the line that caused the problem is also printed, up to the point at which the problem occurred. See Section 3.6 for further details and examples.

3.4.1 Preprocess

The preprocessor reads the DMIS input file and writes the file PrEpRoCeSsDmIs. In general, one line of input becomes one line of output, except that comments and blank lines are deleted, continued lines are concatenated together (with the continuation signs removed), and CALL statements are modified as described below.

The length of every line of the input file is checked. If a line is more than 65536 characters long, the dmisParser prints an error message and quits.

Each line of the input file is checked to be sure there is a carriage return followed by a line feed at the end. If one of those characters is found but the other is missing, an error message is printed, but the missing character is inserted where it belongs and processing continues.

If a comment line (one starting with $$) follows a continued line (one ending with $), an error message is printed, and processing continues.

The line number from the input file is inserted at the beginning of each output line. Where there are blank lines or continuation lines in the input, those line numbers do not appear in the output.

In order to check MACROs, when the preprocessor reads a DMIS MACRO, it reprints the MACRO in the preprocessed file and saves the text of the MACRO. At the point where the MACRO is CALLed, the text of the MACRO is inserted in the preprocessed file with the arguments to the CALL substituted for the MACRO arguments. This creates a callBlock starting with the CALL statement and ending with the ENDMAC statement copied in from the MACRO. For more details, see the in-line documentation in utilityComponents/linuxSun/source/dmis.y of the functions isMacro, isCall, findMacroArgs, findCallArgs, insertCalledMacro, doCall, and doMacro.

3.4.2 Parse

In the parse phase, the dmisParser reads and parses the PrEpRoCeSsDmIs file. While the dmisParser parses, it builds a parse tree in terms of the C++ classes that represent DMIS. Details of parse tree structure are given in the system builders manual.

Some of the error and warning messages reported in the parse phase were written by a programmer and identify a very specific error (such as “variable reused”). Many of the error and warning messages, however, are generated automatically by automatically generated software. These tend to be generic and are often not intuitively clear.

When the dmisParser parses the preprocessed file and reads a DMIS MACRO, the dmisParser makes no attempt to parse it (the dmisParser cannot parse it because the types of the arguments are not specified). But when the dmisParser parses a CALL to the MACRO, it does parse it (in the CALL, the types of the things that replaced the MACRO arguments are known). If an error occurs while a CALLed MACRO is being parsed, the line number given is the line number of the CALL, not the line number of the MACRO.

At the end of the parse phase, two summary messages of the form “N errors” and “M warnings” are printed and the PrEpRoCeSsDmIs file is removed.

3.5 What does the dmisParser do when it parses a set of DMIS input files?

If the first argument to the command to run the dmisParser does not end in .dmi, the dmisParser assumes the file contains a list of names of DMIS input files. The dmisParser handles the files in

the list in the order given and does the following for each.

First, the dmisParser checks that the name ends in .dmi and checks that the file exists. If either of these checks fails, the dmisParser prints an error message and quits.

Then the dmisParser processes the file as described in Section 3.4.

3.6 What do the dmisParser's error and warning messages mean?

If the dmisParser runs into an error during the preprocessing stage, it will print an error message and fix the problem in the preprocessing output file if it can. If the input file or preprocessing output file cannot be opened or if the dmisParser reads a line more than 65536 characters long, it will print an error message and quit. Every error message that can be sent during preprocessing is easy to understand. If there is a preprocessing error, the line number of the line causing the error will be printed, but the line will not be printed. For example the following preprocessor error message means that line 3 is the last line of the file and the endline character is missing from that line.

```
3: Error - no endline on last line of file
```

If the dmisParser runs into an apparent error while parsing the preprocessed file, it will print two lines: (1) the line number from the input file followed by either an error message or a warning message, and (2) the text of the line on which the error occurred, but only up to the point where an error was detected. For example, the following two lines mean that an error was found on line 7, and the error is that a MACRO has two arguments that are the same. The actual line 7 may be longer, but the dmisParser stopped parsing the line when it ran into the error at the second x.

```
7: argument reused
M(featdef)=MACRO/x,x
```

The error message above was written by a human. The error messages written by a human are usually understandable with a little study. Often it will help to refer to the text of the DMIS 5.2 standard.

Many error messages are machine-authored, not written by a human. The machine-authored error messages are generated automatically on the fly by the dmisParser and may be difficult to understand. They all start with "syntax error". Here is a machine-authored example.

```
4: syntax error, unexpected RPAREN, expecting C
y = ASSIGN/MN(x)
```

In this example, the dmisParser has been reading the ASSIGN statement and has just read the right parenthesis (known to the dmisParser as RPAREN) following x. The dmisParser knows that the MN function must have at least two arguments and the arguments are separated by commas (known to the dmisParser as C). The dmisParser was expecting to see a comma after x, but it saw a right parenthesis instead, so it stopped parsing the line and printed an error message.

The dmisParser stops trying to parse a line as soon as it finds an error on the line, but then it tries to continue parsing starting with the next line. Usually this is successful, but sometimes the dmisParser will become confused and start giving off incorrect error messages. This is most likely to happen if an error occurs on the first line of a block (such as in a MEAS or GOTARG statement). To deal with this problem, the error that caused the initial error message should be fixed in the DMIS input file, and the file should be run through the dmisParser again.

Warning messages usually indicate errors. The dmisParser can be tricked into thinking there is a problem for multiply DECLared variables, multiply defined labels, and references to undefined labels even if there is no problem. When it detects a problem of any of those types, since it cannot be sure there is an error, the dmisParser emits a warning message rather than an error message. The percentage of cases in which a warning is not actually an error is very low, however.

4 The dmisConformanceChecker

The dmisConformanceChecker checks that DMIS input files conform to a conformance class of DMIS. It also gathers information on the use of DMIS statements in a set of DMIS input files. The dmisConformanceChecker starts processing a DMIS input file by running the same parser as used in the dmisParser, so the operations described in Section 3.4 and the messages described in Section 3.6 apply to the dmisConformanceChecker. The dmisConformanceChecker does its work by analyzing the parse tree(s) built by the parser.

4.1 How do I use the dmisConformanceChecker to check that a single DMIS input file conforms to a particular DMIS conformance class?

For all three operating systems, you run the dmisConformanceChecker by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

The dmisConformanceChecker takes one or more command arguments. The first argument is either (1) the path to a DMIS input file to parse or (2) the path to a file containing the paths to a number of DMIS input files. If there is only one argument, full DMIS is used as the conformance class. If there are more arguments, they identify the conformance modules in your conformance class. If there are at least two arguments, the second argument must identify a level of an AP. The choices for the second argument are: PM1, PM2, PM3, TW1, TW2, or TW3. Those indicate levels 1, 2, or 3 of the prismatic AP or the thin-walled AP. If there are more than two arguments, the remaining arguments must each identify a level of one of the addenda. The choices are zero or one from each of the following groups of three, in any order: (RY1, RY2, RY3), (MC1, MC2, MC3), (CT1, CT2, CT3), (IP1, IP2, IP3), (QI1, QI2, QI3), (MU1, MU2, MU3), (SF1, SF2, SF3).

The dmisConformanceChecker writes its messages in the command window, but output redirection may be used to send the messages to a file.

4.1.1 Linux and SunOS

Example 1. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceChecker ../../parserTestFiles/okIn/units1.dmi PM3

The command parses the DMIS input file units1.dmi, runs a conformance check, and writes the messages "`0 parser errors`", "`0 parser warnings`", and "`0 conformance checker errors`" on three lines in the command window.

Example 2. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceChecker ../../parserTestFiles/okIn/units1.dmi PM2

The command parses the DMIS input file units1.dmi, runs a conformance check, and writes the following messages. The conformance checker error messages are given because ANGRAD, CM, METER, and FEET, which are in the units1.dmi file, are not allowed in level 2 of the prismatic AP.

```
0 parser errors
0 parser warnings

no angleUnit_ANGRAD subtype of angleUnit
ANGRAD

no lengthUnit_CM subtype of lengthUnit
CM

no lengthUnit_METER subtype of lengthUnit
METER

no lengthUnit_FEET subtype of lengthUnit
FEET

4 conformance checker errors
```

Example 3. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceChecker ../../parserTestFiles/okIn/units1.dmi

The command parses the DMIS input file units1.dmi and writes the messages "`0 parser errors`" and "`0 parser warnings`" on two lines in the command window. This command checks only that the file conforms to full DMIS.

Example 4. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceChecker ../../parserTestFiles/annexAIn/A.07.dmi PM3 QI2 SF1

The command parses the DMIS input file A.07.dmi, runs a conformance check, and writes the messages "`0 parser errors`", "`0 parser warnings`", and "`0 conformance checker errors`" on three lines in the command window. This indicates that level 3 of the prismatic AP, plus level 2 of the QIS addendum and level 1 of the soft gauging addendum are sufficient to handle A.07.dmi. If QI3 had been used instead of QI2 (or SF2 or SF3 instead of SF1), the same messages would have been written.

4.1.2 Windows

Example 1. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceChecker ..\..\parserTestFiles\okIn\units1.dmi PM3

The command parses the DMIS input file units1.dmi, runs a conformance check, and writes the messages "`0 parser errors`", "`0 parser warnings`", and "`0 conformance checker errors`" on three lines in the command window.

Example 2. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceChecker ..\..\parserTestFiles\okIn\units1.dmi PM2

The command parses the DMIS input file units1.dmi, runs a conformance check, and writes the following messages. The conformance checker error messages are given because ANGRAD, CM, METER, and FEET, which are in the units1.dmi file, are not allowed in level 2 of the prismatic AP.

```
0 parser errors
0 parser warnings

no angleUnit_ANGRAD subtype of angleUnit
ANGRAD

no lengthUnit_CM subtype of lengthUnit
CM

no lengthUnit_METER subtype of lengthUnit
METER

no lengthUnit_FEET subtype of lengthUnit
FEET

4 conformance checker errors
```

Example 3. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceChecker ..\..\parserTestFiles\okIn\units1.dmi

The command parses the DMIS input file units1.dmi and writes the messages "`0 parser errors`" and "`0 parser warnings`" on two lines in the command window. This command checks only that the file conforms to full DMIS.

Example 4. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceChecker ..\..\parserTestFiles\annexAIn\A.07.dmi PM3 QI2 SF1

The command parses the DMIS input file A.07.dmi, runs a conformance check, and writes the messages "`0 parser errors`", "`0 parser warnings`", and "`0 conformance checker errors`" on three lines in the command window. This indicates that level 3 of the prismatic AP, plus level 2 of the QIS addendum and level 1 of the soft gauging addendum are sufficient to handle A.07.dmi. If QI3 had been used instead of QI2 (or SF2 or SF3 instead of SF1), the same messages would have been written.

4.2 How can I use a single command to run conformance checks and collect data on statement usage for a whole set of DMIS input files?

Start by making a file listing the names of the DMIS input files you want to test. The file names must include the path. In the examples below, the runAllFull and runSomeFull files are of this sort. The runSomeFull file is about half of the runAllFull file.

4.2.1 Linux and SunOS

To run a set of DMIS input files through the dmisConformanceChecker in Linux, get into the utilities/linux/full directory; in SunOS, get into the utilities/sun/full directory.

Example 1. Give the command:

../bin/dmisConformanceChecker runAllFull

The command runs the 327 DMIS input files listed in the runAllFull file through the dmisConformanceChecker. Output printing goes to the command window. The names of the DMIS files must end in .dmi.

The command takes 5 seconds or so to run on a Dell Precision 670 PC running Linux and about 8 seconds on a Sun Fire V215 running SunOS, and it prints 2602 lines. Most of the lines give the names of each DMIS input file parsed and the parser warnings and parser errors report for each input file. At the end, the following summary report is printed (where many lines have been omitted, as shown by …). The meaning of the summary report should be obvious. There are no names of DMIS statements that were not used at the end because all of the statements were used.

```
Total statement uses for all files
aclrat 28
algdef 4
assign 211
badtst 6
bound 4
calibMaster 1
calibRtab 3
calibSens 12
...
vform 30
windef 5
wkplan 5
wrist 20
write 10
xtern 5
xtract 6
100.0% of the commands in full DMIS were used.
```

Example 2. Give the command:

../bin/dmisConformanceChecker runSomeFull

The command runs the 143 DMIS input files listed in the runSomeFull file through the dmisConformanceChecker. Output printing goes to the command window. The names of the DMIS files must end in .dmi.

The command takes 2 seconds or so to run on a Dell Precision 670 PC running Linux and about 3 seconds on a Sun Fire V215 running SunOS, and it prints 1216 lines. Most of the lines give the names of each DMIS input file parsed and the parser errors and parser warnings report for each input file. At the end, the following summary report is printed (where many lines have been

omitted, as shown by …).

```
Total statement uses for all files
aclrat 27
algdef 2
assign 51
badtst 4
bound 4
calibMaster 1
...
units 2
vform 4
wrist 3
xtern 4
67.9% of the commands in full DMIS were used.
The following commands were not used:
lotid
operid
pameas
partid
partrv
...
uncertset
value
windef
wkplan
write
xtract
```

Example 3. Give the command:

../bin/dmisConformanceChecker runSomeFull PM3 RY3 MC3 CT3 IP3 QI3 MU3 SF3

The command runs the 143 DMIS input files listed in the runSomeFull file through the dmisConformanceChecker. The conformance class being used includes everything that is allowed in level 3 of the prismatic AP and all 7 addenda. Output printing goes to the command window. The names of the DMIS files must end in .dmi.

The command takes 2 seconds or so to run on a Dell Precision 670 PC running Linux and about 3 seconds on a Sun Fire V215 running SunOS, and it prints 1905 lines. Most of the lines give the names of each DMIS input file parsed and the parser warnings, parser errors, and conformance errors report for each input file. At the end, the following summary report is printed (where many lines have been omitted, as shown by …). The meaning of the summary report should be obvious except for the `M` at the beginning of some lines. The "Total statement uses" list at the top includes only statements that are in the conformance class. An `M` at the beginning of a line means that the statement is essential for the metrology functionality of the conformance class. In the example, for instance, "`M pameas`" occurs in the second section of the report because pameas is essential in the CT3 (contact scanning level 3) conformance module, but no pameas command was used in any of the input files. The `M` is used only in the first and second sections of the report.

```
Total statement uses for all files
aclrat 27
assign 51
badtst 4
bound 4
calibMaster 1
...
M tolProfp 1
M tolProfs 1
tooldf 2
units 2
vform 4
wrist 3
xtern 4
67.4% of the commands in the conformance class were used.
The following commands were not used:
lotid
operid
M pameas
partid
partrv
...
M tolWidth
M trans
uncertalg
uuncertset
value
wkplan
write
xtract
The following commands not in the conformance class were used:
algdef
croscl
dmisOff
dmisOn
featEdgept
featGeom
featObject
fildef
geom
group
litdef
refmnt
```

4.2.2 Windows

To run a set of DMIS input files through the dmisParser in Windows, get into the utilities\windows\full directory.

Example 1. Give the command:

..\bin\dmisConformanceChecker.exe runAllFull

The command runs the 327 DMIS input files listed in the runAllFull file through the dmisParser. Output printing goes to the command window. The names of the DMIS files must end in .dmi. In Windows, it does not matter whether or not you include the .exe suffix in the name of the command.

The command takes about 6 seconds to run on a Dell Dimension 8300 PC running Windows XP, and it prints 2602 lines. Most of the lines give the names of each DMIS input file parsed and the parser warnings and parser errors for each input file. At the end, the summary report shown in Example 1 of Section 4.2.1 is printed (where many lines have been omitted, as shown by …). The meaning of the summary report should be obvious. There are no names of DMIS statements that were not used at the end because all of the statements were used.

Example 2. Give the command:

..\bin\dmisConformanceChecker runSomeFull

The command runs the 143 DMIS input files listed in the runSomeFull file through the dmisConformanceChecker. Output printing goes to the command window. The names of the DMIS files must end in .dmi.

The command takes about 3 seconds to run on a Dell Dimension 8300 PC running Windows XP, and it prints 1216 lines. Most of the lines give the names of each DMIS input file parsed and the parser errors and parser warnings report for each input file. At the end, the summary report shown in Example 2 of Section 4.2.1 is printed (where many lines have been omitted, as shown by …).

Example 3. Give the command:

..\bin\dmisConformanceChecker runSomeFull PM3 RY3 MC3 CT3 IP3 QI3 MU3 SF3

The command runs the 143 DMIS input files listed in the runSomeFull file through the dmisConformanceChecker. The conformance class being used includes everything that is allowed in level 3 of the prismatic AP and all 7 addenda. Output printing goes to the command window. The names of the DMIS files must end in .dmi.

The command takes about 3 seconds to run on a Dell Dimension 8300 PC running Windows XP, and it prints 1905 lines. Most of the lines give the names of each DMIS input file parsed and the parser warnings, parser errors, and conformance errors report for each input file. At the end, the summary report shown and explained in Example 3 of Section 4.2.1 is printed.

5 The dmisConformanceRecorder

The dmisConformanceRecorder automatically inserts conformance information on the `DMISMN` or `DMISMD` line of a DMIS input file. The conformance information identifies which conformance modules are needed to deal with the file. The dmisConformanceRecorder starts processing a DMIS input file by running the same parser as used in the dmisParser, so the operations described in Section 3.4 and the messages described in Section 3.6 apply to the dmisConformanceRecorder. The dmisConformanceRecorder does its work by analyzing the parse tree built by the parser.

5.1 How do I use the dmisConformanceRecorder to put conformance information into a DMIS input file?

For all three operating systems, you run the dmisConformanceRecorder by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

The dmisConformanceRecorder takes one to four command arguments. The first argument is the path to a DMIS input file in which to insert conformance information. The other arguments may be any or all of PM, TW, and QI in any order.

If neither PM nor TW is given or if both are given, then the conformance information will include both PM,*n* and TW,*m* where *m* and *n* are each 1, 2, 3, or 4. If only PM is given, then the conformance information will include only PM,*n*. If only TW is given, then the conformance information will include only TW,*m*.

For some DMIS input files, the dmisConformanceRecorder will have a choice between IP and QI because several DMIS statements are included in both addenda at the same level. For such files, if QI is not given as an argument, the dmisConformanceRecorder will choose to use IP since IP has fewer requirements than QI. If QI is given as an argument and there is a choice, the dmisConformanceRecorder will choose to use QI.

If there is a parse error in the file, error messages will be printed and no conformance statement will be generated. The file will not change.

5.1.1 Linux and SunOS

Example 1. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following commands.

cp ../../parserTestFiles/annexAIn/A.07.dmi junk

bin/dmisConformanceRecorder junk

The command will run almost instantly, and the following messages will be printed.

```
running command "dmisConformanceRecorder junk"
starting NIST DMIS Conformance Recorder, Version 2.2.1
for DMIS Version 5.2
parsing file junk
parsed all 58 lines of file - no parser errors
conformance statement "PM,3, TW,3, QI,2, SF,1" inserted at end of
DMISMN line
original file copied to junk.back
exiting NIST DMIS Conformance Recorder
```

The junk file will not change, so if you are feeling skeptical, edit junk by deleting everything after `5.2` on the `DMISMN` line of junk before running the dmisConformanceRecorder. Then look at junk again afterwards.

Example 2. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following commands.

cp ../../parserTestFiles/annexAIn/A.07.dmi junk

bin/dmisConformanceRecorder junk TW

The same messages shown in the previous example will be printed except that "PM,3" will not be in the conformance statement. After the dmisConformanceRecorder finishes running, compare the first line of A.07.dmi with the first line of junk. They will be identical except that PM,3 will be missing from junk.

Example 3. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following commands.

cp ../../parserTestFiles/okIn/clmpid1.dmi junk

bin/dmisConformanceRecorder junk QI

After these commands are given, compare the DMISMN line of clmpid1.dmi with the DMISMN line of junk. They will be the following:

```
DMISMN / 'DMIS parser test for CLMPID', 05.2, PM,1, TW,1, IP,2
DMISMN / 'DMIS parser test for CLMPID', 05.2, PM,1, TW,1, QI,2
```

Note that the last thing on the line has changed from IP,2 to QI,2. Either IP2 or QI2 is sufficient to handle this file. Since IP is preferred to QI by default, it was used in clmpid1.dmi. Since QI appeared at the end of the dmisConformanceRecorder command, QI was used in junk.

5.1.2 Windows

Example 1. Get into the utilities\windows directory. Give the following commands.

copy ..\..\parserTestFiles\annexAIn\A.07.dmi junk

bin\dmisConformanceRecorder junk

The command will run almost instantly, and the following messages will be printed.

```
running command "dmisConformanceRecorder junk"
starting NIST DMIS Conformance Recorder, Version 2.2.1
for DMIS Version 5.2
parsing file junk
parsed all 58 lines of file - no parser errors
conformance statement "PM,3, TW,3, QI,2, SF,1" inserted at end of
DMISMN line
original file copied to junk.back
exiting NIST DMIS Conformance Recorder
```

The junk file will not change, so if you are feeling skeptical, edit junk by deleting everything after 5.2 on the DMISMN line of junk before running the dmisConformanceRecorder. Then look at junk again afterwards.

Example 2. Get into the utilities\windows directory. Give the following commands.

copy ..\..\parserTestFiles\annexAIn\A.07.dmi junk

bin\dmisConformanceRecorder junk TW

The same messages shown in the previous example will be printed except that "PM,3" will not be in the conformance statement. After the dmisConformanceRecorder finishes running, compare the first line of A.07.dmi with the first line of junk. They will be identical except that PM,3 will

be missing from junk.

Example 3. Get into the utilities\windows directory. Give the following commands.

copy ..\..\parserTestFiles\okIn\clmpid1.dmi junk

bin\dmisConformanceRecorder junk QI

After these commands are given, compare the `DMISMN` line of clmpid1.dmi with the `DMISMN` line of junk. They will be the following:

```
DMISMN / 'DMIS parser test for CLMPID', 05.2, PM,1, TW,1, IP,2
DMISMN / 'DMIS parser test for CLMPID', 05.2, PM,1, TW,1, QI,2
```

Note that the last thing on the line has changed from `IP,2` to `QI,2`. Either IP2 or QI2 is sufficient to handle this file. Since IP is preferred to QI by default, it was used in clmpid1.dmi. Since QI appeared at the end of the dmisConformanceRecorder command, QI was used in junk.

5.2 What does the conformance information mean?

If both TW and PM are given in the conformance information, they are alternatives, but if any of the addenda are included, they are all required. For example, `PM,3, TW,3, QI,2, SF,1` means that either PM3, QI2, and SF1 are required or TW3, QI2, and SF1 are required.

A 1, 2, or 3 after PM, TW, or any of the addenda initials indicates that level 1, 2, or 3 of that AP or addendum is needed. A 4 may appear after PM and TW but not after any of the addenda initials. A 4 after PM indicates that the file includes a construct that is in full DMIS but not in any level of PM and not in any level of any addendum. A 4 after TW indicates that the file includes a construct that is in full DMIS but not in any level of TW and not in any level of any addendum.

The level of each conformance module chosen by the dmisConformanceRecorder is the minimum level required to handle the file.

6 The dmisConformanceTester

Running the dmisConformanceTester on a DMIS input file is similar to first running the dmisConformanceChecker and then running the dmisConformanceRecorder. There are two differences in the second half of the action. First, the dmisConformanceTester prints conformance information in the command window rather than inserting it in a file. Second, the dmisConformanceTester prints the C++ construct that first caused each conformance module to have the level it has.

6.1 How do I use the dmisConformanceTester?

For all three operating systems, you run the dmisConformanceTester by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

The dmisConformanceTester takes two to nine arguments. The first argument is the path to a DMIS input file to test. The second argument is a level of an AP (one of PM1, PM2, PM3, TW1, TW2, or TW3). Any additional arguments are each a level of an addendum, i.e., 0 or 1 from each of the following 7 sets (RY1,RY2,RY3), (CT1,CT2,CT3), (MC1,MC2,MC3), (IP1,IP2,IP3) (QI1,QI2,QI3), (MU1,MU2,MU3), (SF1,SF2,SF3).

If the dmisConformanceTester has a choice between IP and QI, it will choose IP unless the arguments include QI and do not include IP.

6.1.1 Linux and SunOS

Example 1. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceTester ../../parserTestFiles/okIn/clmpid1.dmi PM1 QI1

The following messages will be printed in the command window. The first five lines are the same as what the dmisConformanceChecker would print. The `CLMPID` statement is in QI2 but not in QI1, so a conformance error message is given. The sixth line is what the dmisConformanceRecorder would add to the `DMISMN` line of the clmpid1.dmi file. The last two lines are explained in Section 6.2.

```
0 parser errors
0 parser warnings

no clmpidStm subtype of dmisFreeStatement
CI(C1)=CLMPID/’eclampsia’

1 conformance checker error

PM,1, QI,2
PM1 inputFile a_dmisFirstStatement
QI2 clmpidStm
```

Example 2. Get into the utilities/linux directory if you are using Linux or the utilities/sun directory if you are using SunOS. Give the following command.

bin/dmisConformanceTester ../../parserTestFiles/okIn/clmpid1.dmi PM1 QI2

The following messages will be printed in the command window. The first three lines are the same as what the dmisConformanceChecker would print. The `CLMPID` statement is in QI2, so there are no conformance errors. The fourth line is what the dmisConformanceRecorder would add to the `DMISMN` line of the clmpid1.dmi file. The last two lines are explained in Section 6.2.

```
0 parser errors
0 parser warnings

0 conformance checker errors

PM,1, QI,2
PM1 inputFile a_dmisFirstStatement
QI2 clmpidStm
```

6.1.2 Windows

Example 1. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceTester ..\..\parserTestFiles\okIn\clmpid1.dmi PM1 QI1

The following messages will be printed in the command window. The first five lines are the same as what the dmisConformanceChecker would print. The `CLMPID` statement is in QI2 but not in QI1, so a conformance error message is given. The sixth line is what the dmisConformanceRecorder would add to the `DMISMN` line of the clmpid1.dmi file. The last two lines are explained in Section 6.2.

```
0 parser errors
0 parser warnings

no clmpidStm subtype of dmisFreeStatement
CI(C1)=CLMPID/’eclampsia’

1 conformance checker error

PM,1, QI,2
PM1 inputFile a_dmisFirstStatement
QI2 clmpidStm
```

Example 2. Get into the utilities\windows directory. Give the following command.

bin\dmisConformanceTester ..\..\parserTestFiles\okIn\clmpid1.dmi PM1 QI2

The following messages will be printed in the command window. The first three lines are the same as what the dmisConformanceChecker would print. The `CLMPID` statement is in QI2, so there are no conformance errors. The fourth line is what the dmisConformanceRecorder would add to the `DMISMN` line of the clmpid1.dmi file. The last two lines are explained in Section 6.2.

```
0 parser errors
0 parser warnings

0 conformance checker errors

PM,1, QI,2
PM1 inputFile a_dmisFirstStatement
QI2 clmpidStm
```

6.2 What does the output of the dmisConformanceTester mean?

The beginning of the output from the dmisConformanceTester (through the line giving the number of conformance checker errors) is the same as the output produced by the dmisConformanceChecker. See Section 4.1.1 or Section 4.1.2.

The next line of output from the dmisConformanceTester is a listing of conformance information in the same format as produced by the dmisConformanceRecorder. See Section 5.2.

The remaining lines of output show the C++ construct first encountered by the dmisConformanceTester that caused each conformance module to have the level it has. For example, suppose the lines are:

```
PM1 inputFile a_dmisFirstStatement
IP2 clmpidStm
```

The first line means that the dmisConformanceTester decided prismatic level 1 is required as soon as it started traversing the parse tree (which has `inputFile` at its root). In order to handle the `a_dmisFirstStatement` attribute of the `inputFile` class when the prismatic AP is being used, level 1 is required. Whenever a message has three parts, as on the first line, the third item is an attribute of the C++ class named in the second item.

The second line means that when the dmisConformanceTester encountered an instance of the `clmpidStm` C++ class, it decided that level 2 of the IP addendum is required. Whenever a message has two parts, as on the second line, the second item is the name of a C++ class.

The intent of providing this information is to help the user understand why the particular levels of particular conformance modules are selected. It would be clearer if sections of DMIS code were given in addition to the C++ class information, but implementing that has not yet been attempted. In most cases (such as the second line above), the meaning of the message will be easy to understand since the names of the C++ classes and their attributes have been chosen to be similar to the DMIS names.

There may be many items in a DMIS input file, each of which forces a conformance module to the same maximum level. The messages from the dmisConformanceTester only identify the first of those items encountered.

7 The dmisTestFileReductor

7.1 What does the dmisTestFileReductor do?

The dmisTestFileReductor provides an easy way to generate a set of parser test files for any of the 98,304 allowed combinations of conformance modules[1]. In its simplest use, the dmisTestFileReductor takes a marked DMIS input file (the incoming file) as input and produces a similar but reduced DMIS input file (the outgoing file) as output. The reduction is made by removing some lines entirely. Which lines of the incoming file are removed is determined by which conformance module names are used as arguments when the dmisTestFileReductor is called. In some cases, no outgoing file is produced. An incoming file may be identified by using its name as an argument when the dmisTestFileReductor is called. If the name of a file containing a list of the names of DMIS input files is used as an argument instead of the name of a DMIS input file, then the entire list of DMIS input files will be processed.

The name of the outgoing file is the same as the name of the incoming file, but the outgoing file is placed in a directory whose name is another argument to the call to the dmisTestFileReductor.

1. There are 6 choices of AP level, and 4 choices (not at all, 1, 2, or 3) for each of 7 addenda (6x4x4x4x4x4x4x4 = 98,304).

7.2 How are incoming files marked?

The files in the parserTestFiles/okIn directory are marked for use with the dmisTestFileReductor. Each of those files is designed to test one particular DMIS statement, and the file name contains the name of that statement. No other files in the test suite are appropriate to use as incoming files. As described in Section 8, the parser test files are syntactically correct but may not make sense as programs. The files outgoing from the dmisTestFileReductor will make even less sense, so they should be used only for testing parsers.

Each DMIS code line of each of the 254 DMIS input files in the parserTestFiles/okIn directory is preceded by a comment line naming the minimum conformance modules needed to handle the DMIS code line. In addition, the first line of each file names the minimum conformance modules needed to make the file worth using.

For example, here is a file that could be used for testing handling of the FEDRAT statement.

```
$$ PM2 TW2
$$ PM1 TW1
DMISMN / 'DMIS parser test for FEDRAT', 05.2, PM,2, TW,3, RY,2
$$ PM2 TW2
FEDRAT/POSVEL,IPS,19.5
$$ PM2 TW3
FEDRAT/MESVEL,MPM,19.5
$$ RY2
FEDRAT/ROTVEL,RPM,19.5
$$ PM1 TW1
ENDFIL
```

The first line is "`$$ PM2 TW2`". That means that in order to be worth using this file for testing the FEDRAT statement, at least level 2 of the prismatic AP or level 2 of the thin-walled AP must be used. If the arguments to the dmisTestFileReductor include only one conformance module name and that name is PM1 or TW1, then no outgoing file will be produced. If the arguments include any of PM2, PM3, TW2, or TW3, then an outgoing file will be produced.

The second line is "`$$ PM1 TW1`" because either level 1 of the prismatic AP or level 1 of the thin-walled AP is adequate for handling the DMISMN statement on the third line.

The fourth line is "`$$ PM2 TW2`" because at least level 2 of the prismatic AP or level 2 of the thin-walled AP is adequate for handling the FEDRAT/POSVEL statement on the fifth line.

The sixth line is "`$$ PM2 TW3`" because at least level 2 of the prismatic AP or level 3 of the thin-walled AP is adequate for handling the FEDRAT/MESVEL statement on the seventh line.

The eighth line is "`$$ RY2`" because at least level 2 of the rotary table addendum is needed for handling the FEDRAT/ROTVEL statement on the ninth line.

In the markings, any PM and TW levels are alternatives, but if other conformance module names are given, they are required.

To avoid parse errors in reduced files, where there is a block of statements in a file, the markings on the lines inside the block may be changed from what they would normally be so that the entire block is deleted if the statement starting the block is deleted. For the same reason, lines referencing a label may be marked the same as the line that defined the label.

7.3 How do I use the dmisTestFileReductor?

For all three operating systems, you run the dmisTestFileReductor by typing a command in a command window. The same procedure is used for Linux and SunOS. Windows is slightly different.

The dmisTestFileReductor takes three to ten arguments. The first argument is the path to a DMIS input file to test or to a file containing a list of names of DMIS input files to test. The second argument is the path to a directory in which to write the outgoing file(s). The third argument is a level of an AP (one of PM1, PM2, PM3, TW1, TW2, or TW3). Any additional arguments are each a level of an addendum, i.e., 0 or 1 from each of the following 7 sets (RY1,RY2,RY3), (CT1,CT2,CT3), (MC1,MC2,MC3), (IP1,IP2,IP3) (QI1,QI2,QI3), (MU1,MU2,MU3), (SF1,SF2,SF3).

7.3.1 Linux and SunOS

Example 1. Get into the utilities/linux/full directory if you are using Linux or the utilities/sun/full directory if you are using SunOS. Give the following command.

../bin/dmisTestFileReductor ../../../parserTestFiles/okIn/fedrat1.dmi outgoing PM2

No messages will be printed in the command window. After the command runs, a file named fedrat1.dmi will be in the outgoing subdirectory. That file will contain only those lines of DMIS code from the original fedrat1.dmi that can be executed by a system that implements level 2 of the prismatic AP. The comment lines immediately preceding those DMIS code lines will also be in the new version of the file. In addition, the requirements at the end of the `DMISMN` line will be updated.

Example 2. Get into the utilities/linux/full directory if you are using Linux or the utilities/sun/full directory if you are using SunOS. Give the following command.

../bin/dmisTestFileReductor ../../../parserTestFiles/okIn/fedrat1.dmi outgoing TW2 RY2

No messages will be printed in the command window. After the command runs, a file named fedrat1.dmi will be in the outgoing subdirectory. That file will contain only those lines of DMIS code from the original fedrat1.dmi that can be executed by a system that implements both level 2 of the thin-walled AP and level 2 of the rotary table addendum. The comment lines immediately preceding those DMIS code lines will also be in the new version of the file. In addition, the requirements at the end of the `DMISMN` line will be updated.

Example 3. Get into the utilities/linux/full directory if you are using Linux or the utilities/sun/full directory if you are using SunOS. Remove any files that may be in the outgoing subdirectory. Give the following command.

../bin/dmisTestFileReductor runOkFull outgoing PM2

The runOkFull file contains the paths to all the DMIS input files in the parserTestFiles/okIn directory. While the command runs, a number of messages will be printed in the command window, each saying that some file is not being generated “because file requirements did not meet conformance class requirements”. After the command is finished running, a set of 136 new parser files will have been printed in the outgoing subdirectory. Each file will be reduced as described in Example 1. This set of files should be useful for testing a parser that implements only level 2 of the prismatic AP.

7.3.2 Windows

Example 1. Get into the utilities\windows\full directory. Give the following command.

..\bin\dmisTestFileReductor ..\..\..\parserTestFiles\okIn\fedrat1.dmi outgoing PM2

No messages will be printed in the command window. After the command runs, a file named fedrat1.dmi will be in the outgoing subdirectory. That file will contain only those lines of DMIS code from the original fedrat1.dmi that can be executed by a system that implements level 2 of the prismatic AP. The comment lines immediately preceding those DMIS code lines will also be in the new version of the file. In addition, the requirements at the end of the `DMISMN` line will be updated.

Example 2. Get into the utilities\windows\full directory. Give the following command.

..\bin\dmisTestFileReductor ..\..\..\parserTestFiles\okIn\fedrat1.dmi outgoing TW2 RY2

No messages will be printed in the command window. After the command runs, a file named fedrat1.dmi will be in the outgoing subdirectory. That file will contain only those lines of DMIS code from the original fedrat1.dmi that can be executed by a system that implements both level 2 of the thin-walled AP and level 2 of the rotary table addendum. The comment lines immediately preceding those DMIS code lines will also be in the new version of the file. In addition, the requirements at the end of the `DMISMN` line will be updated.

Example 3. Get into the utilities\windows\full directory. Remove any files that may be in the outgoing subdirectory. Give the following command.

..\bin\dmisTestFileReductor runOkFull outgoing PM2

The runOkFull file contains the paths to all the DMIS input files in the parserTestFiles/okIn directory. While the command runs, a number of messages will be printed in the command window, each saying that some file is not being generated “because file requirements did not meet conformance class requirements”. After the command is finished running, a set of 136 new parser files will have been printed in the outgoing subdirectory. Each file will be reduced as described in Example 1. This set of files should be useful for testing a parser that implements only level 2 of the prismatic AP.

8 Parser Test Files

Note: In general, parser test files will cause errors in DMIS execution systems. Use files from the systemsTestFiles directory (see Section 9) if you want to test a DMIS execution system.

8.1 What are the parser test files like?

The files in the parserTestFiles directory conform to the syntax rules implicit in the EBNF for DMIS plus additional rules about variables and labels.

The syntax rules implicit in the EBNF include the syntax described in the “Input Formats” descriptions in section 6 of DMIS 5.2, plus the following:

- Integer values (not real values) must be used where DMIS requires integer values.

The additional rules stated in the text of DMIS 5.2 to which the parser test files conform are:

- All variables must be explicitly declared with `DECL` or implicitly declared by being parameters of a `MACRO`.

- A variable of a given name may not be declared twice with `DECL` or used twice as a parameter of the same `MACRO`.
- Wherever a statement or an expression requires a variable to be of a given type, the variable used must be of the correct type.
- Blocks of a given type may contain only those types of statements that are allowed in blocks of that type.
- All labels must be defined before they are referenced.
- A label of a given type may be defined only once, except for feature labels, since DMIS allows feature labels to be redefined.

Most of the files in the parserTestFiles directory are **not** suitable as test files for DMIS execution systems (systems that carry out the statements in a DMIS input file) because the files:

- may contain semantic errors, such as having a negative number where a positive number is required,
- may contain nonsense such as attempting to measure points on a feature that are not near the feature,
- may crash the equipment.

8.2 What do the “dmi” and “out” filename suffixes mean?

Most of the files in the parserTestFiles directory are DMIS input files and have the suffix .dmi. These are files for testing DMIS utilities.

The OK.out file contains two lines: “`0 errors`” and “`0 warnings`”. All the other files in the parserTestFiles directory that have the suffix .out are files containing the error and warning messages that the NIST DMIS parser prints when it parses a file with the same base name and a .dmi suffix. The files in this directory that have the suffix .out are **not** DMIS output files as defined in DMIS 5.2.

8.3 What is in the parserTestFiles directory?

As shown in Figure 4 and described below, the parserTestFiles directory has six subdirectories.

1. The annexAIn directory contains the examples from Annex A of DMIS 5.2. In most cases these have been modified by adding `DMISMN` and `ENDFIL`.

2. The annexAOut directory contains only one file, A.28.out, which contains messages printed by the dmisParser when it parses A.28.dmi.

3. The errorIn directory contains parser test files with errors in them. The file names of all of these files include the string Error.

4. The errorOut directory contains .out files corresponding to the files in the errorIn directory. Each file contains the error and warning messages printed by the dmisParser when it parses the corresponding file from the corresponding errorIn directory.

5. The okIn directory contains 254 syntactically correct parser test files, with one or more for each subsection of section 6 of DMIS 5.2. For most of these files, the name of the file corresponds to the name of a DMIS statement. These directories also contain syntactically correct programs testing expressions and the preprocessor of the dmisParser. As described in Section 7, each line of DMIS code in each file in this directory is preceded by a comment line that names the minimum conformance modules needed in order for the line to be in conformance. The first line of each file

is a comment naming the minimum conformance modules needed to make the file worth using as a test file for the DMIS statement in the name of the file.

6. The okOut directory contains three files: OK.out (which contains the messages produced by the dmisParser when there are no parser errors or warnings), OKconf.out (which contains the messages produced by the dmisConformanceChecker when there are no parser errors or warnings and no conformance errors), and confusing.out (which contains what the dmisParser will print when it parses the confusing.dmi file).

Figure 4. ParserTestFiles Directory Structure

8.4 For what purposes may the parser test files be used?

The parser test files may be used by developers building DMIS parsers to check that their parsers are working correctly. A method of automating this testing is described in Section 3.3. A method of using the parser test files for full DMIS to produce a set of test files for any conformance class defined by conformance modules is described in Section 7.

The parser test files have been used at NIST for:

- debugging the NIST DMIS utilities
- checking that the 27 files defining the conformance modules for C++ are correct.

See Section 3.1 and Section 3.2 for descriptions of how to use test files to test utilities.

9 System Test Files

9.1 What is in the systemTestFiles directory?

The systemTestFiles directory contains DMIS input files that may be safely run on a DMIS execution system and should run without error.

As shown in Figure 5, the systemTestFiles directory has four subdirectories: full, prismatic1, prismatic2, and prismatic3. Each of them has two subdirectories, one for programs that produce no motion, and one for programs that produce motion.

Each okInNoMotion… subdirectory contains a subset of the files in the corresponding okIn… directory under the parserTestFiles directory (possibly modified). The subset consists of files which do not move the axes and do not contain semantic errors.

The okInMotion… subdirectories contains files that move the sensor. These files contain instructions about how to use themselves, including suggestions for editing. The files in these

directories should only be run by a person who is knowledgeable about using the machine on which they are to be run.

The full/okInMotionFull subdirectory contains the files freeMotion.dmi, gohome.dmi, IMTS1.dmi, and simple1.dmi. The first two of these move the sensor without attempting to inspect anything. Each of the last two inspects a specific machined part, which must be available in order to execute the file.

The prismatic3/okInMotionP3 subdirectory contains the files freeMotionp3.dmi, gohome.dmi, IMTS1.dmi, and simple1.dmi.

The prismatic2/okInMotionP2 subdirectory contains the files freeMotionp2.dmi, IMTS1p2.dmi, and simple1p2.dmi.

The prismatic1/okInMotionP1 subdirectory contains only the file simple1p1.dmi.

A p1, p2, or p3 in a file name before .dmi means the file has been modified to be suitable for the prismatic1, prismatic2, or prismatic3 conformance class.

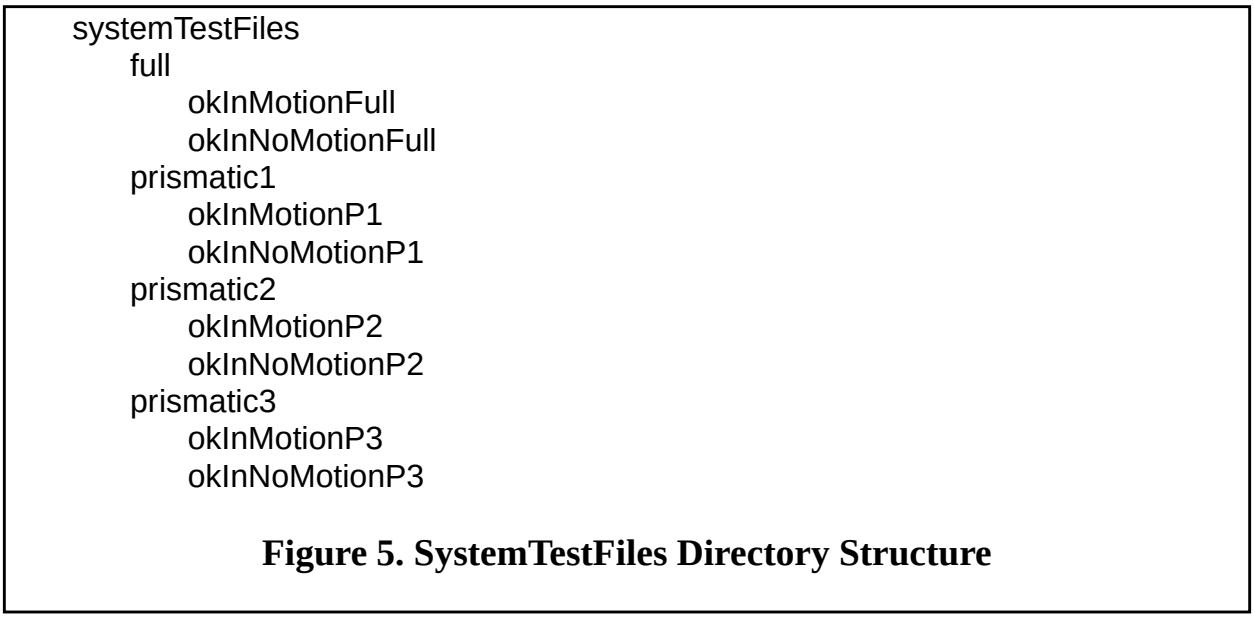

Figure 5. SystemTestFiles Directory Structure

9.2 For what purposes may system test files be used?

System test files may be used by users, conformance testers, or developers to determine if a DMIS execution system can parse and execute DMIS input files correctly. Although many files are provided, they do not cover all of the functionality of a DMIS execution system. Moreover, the DMIS output files that should be generated when these input files are executed are not included in the test suite.

10 EBNF

10.1 Who should read this section?

Read this section if you:
- are building a DMIS parser by modifying a DEBNF file,
- want to deal with a DEBNF file for some other reason,
- are curious about DEBNF.

10.2 What is EBNF?

EBNF (Extended Backus-Naur Form) is an international standard language for describing the syntax of formal languages. EBNF is ISO/IEC standard 14977. A copy of the final standard may be downloaded free of charge from
http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.
The standard gives a good intuitive description of the EBNF language.

10.3 Why use EBNF?

It is a good idea to use a formal syntax language such as EBNF to describe a computer-readable language such as DMIS because a formal language allows the syntax of the target language (DMIS) to be specified completely and unambiguously. It is difficult to give a complete and unambiguous description of syntax any other way. It is a good idea to use a *standard* formal syntax language because being a standard ensures the language has been carefully developed and is publicly available to all.

10.4 What is DEBNF?

DEBNF (DMIS EBNF) is a dialect of EBNF used in the DMIS standard to define the syntax of DMIS. All of the semantics (meanings) of DEBNF are consistent with those of EBNF, but DEBNF uses some conventions that provide a shorthand allowing it to provide additional meanings that could be expressed in EBNF but would require many more pages.

DEBNF uses some, but not all, of the extensions to BNF provided by EBNF.

To be consistent with the DMIS standard, DEBNF is used in the NIST DMIS test suite.

10.5 What is DEBNF Syntax?

10.5.1 Overview

A DEBNF file is a list of productions. Each production sets a production name to be equivalent to a list of definitions. Each definition is a list of expressions. An expression may be (among other things) the name of a production, a token, a single character, or an optional. A production name is also called a non-terminal symbol.

For example, the DEBNF production for the `DMISMN` statement is:

```
dismnStm = DMISMN, '/', stringConst, c, versionTag,
   [c, conformItemList], # ;
```

In this example:
- The name of the production is **`dismnStm`**.

- **=** is an assignment symbol equating the name on the left to the definition on the right.
- There is only one definition, and it has six expressions.
- **DMISMN** is a token.
- **'//'** is a slash.
- **c** is a the name of a production that represents a comma.
- **stringConst** and **versionTag** are the names of other productions.
- **[c, conformItemList]** is an optional consisting of a comma followed by the name of a production.
- **#** is an end-of-line symbol.
- The expressions are separated by commas, and the production is ended by a semicolon.

The production means that a **dmismnStm** is equivalent to the token **DMISMN** followed by a slash followed by a **stringConst** followed by a comma followed by a **versionTag** followed by an optional **conformItemList** preceded by a comma followed by an end-of-line.

The order in which productions are given in a DEBNF file is irrelevant except that the top-level production (**inputFile** for DMIS) must be given first. The order in which the alternate definitions of a production is given is also irrelevant. The ordering of the expressions in a definition, however, is significant. It is OK if a production has no definitions, and it is OK if there are no expressions in a definition.

10.5.2 Rules of Standard EBNF

The following are the elements of standard EBNF used in DEBNF. Standard EBNF includes additional elements that are not needed (and, hence, not used) in DEBNF.

1. An EBNF file is a list of productions in which every production name except the first one is used in defining some other production.

2. A production consists of a non-terminal symbol followed by an equal sign, followed by a (possibly empty) list of alternative definitions, followed by a semicolon.

3. A definition is a list of expressions separated by commas.

4. A vertical bar **|** is used between the definitions of a production. For example, the following means **strVar6** may be either the token **LONG** or the token **SHORT**:

```
strVar6 = LONG | SHORT ;
```

5. An expression is a symbol name (token or non-terminal), a single character, a group, a constant, or an optional.

6. Symbol names start with a letter and include only letters and digits.

7. A single character must be preceded and followed by an apostrophe, e.g., **'//'** .

8. A group is two alternatives enclosed in parentheses (other types of groups are allowed in full EBNF). The alternatives are separated by a vertical bar. For example, the spelling of the token AND is defined as follows in the DEBNF file for full DMIS.

```
AND = '.' , ('A'|'a') , ('N'|'n') , ('D'|'d') , '.' ;
```

This means the spelling may be any of **.AND.** , **.ANd.** , **.AnD.** , **.And.** , **.aND.** , **.aNd.** , **.anD.** , **.and.** .

9. A constant is any string of printable characters or space surrounded by apostrophes, for example **'**'** or **'Not defined here'**.

10. A simple optional expression in a production is set off using square brackets.
For example, **a** , **[b** , **c]** , **d** means that either **a,d** or **a,b,c,d** is allowed.
Simple optional expressions (and multiple optional expressions, which follow) may be nested.
For example, **a , [b ,[c]] , d** means **a,d** or **a,b,d** or **a,b,c,d** is allowed.

11. A multiple optional expression in a production is set off using square brackets preceded by a digit (full EBNF allows any positive integer) and an asterisk. The digit gives the upper limit on the number of repetitions. For example, **a , 2*[b , c] , d** means **a,d** or **a,b,c,d** or **a,b,c,b,c,d** is allowed.

12. White space (spaces, tabs, and newlines) may be used anywhere except inside symbol names and constants and has no meaning. Thus, a single definition may extend across several lines. Spaces (but not tabs or newlines) may be used inside a constant, where they are part of the constant.

13. Comments are indicated by being enclosed with **(*** at the beginning and ***)** at the end. Multiple comments on the same line are allowed and may occur in the middle of definitions. For example, the following is allowed: **a** , **b** , **(* comment1 *) c** , **(* comment2 *) d**. Comments that extend across several lines are also allowed, but comments may not be nested. Comments have no formal meaning. They are treated like white space.

10.5.3 Conventions of DEBNF

DEBNF uses the following conventions in addition to the rules for standard EBNF.

1. Token - A word in upper case letters is a token, e.g., **DMISMN**. In a DMIS input file, the word must appear using the same characters as in the DEBNF file, with the following exceptions. First, in the DMIS file, either lower case or upper case letters may be used (and mixed, e.g., **DmisMn**). Second, since the EBNF standard requires that a symbol name start with a letter and include only letters and digits, DMIS tokens that start with a dot, digit, or minus sign or contain an underscore have been spelled differently. The new spellings are given as productions near the beginning of the DEBNF file. If a token appears on the left side of a production, the right side gives its spelling.

2. The name of a production other than a production spelling a token must start with a lower case letter (digits are also allowed, but underscores are not allowed), except as provided in the next paragraph.

3. The name of a data type that is not meaningfully defined by a production must start with an upper case letter and contain a least one lower case letter, e.g., **StringVarname**. The characters to be used for these data types in a DMIS input file are specified in section 5 of DMIS 5.2. Because the definitions of these data types in terms of characters are complex, these data types are given syntactically correct but meaningless definitions in the DEBNF file, and the software that analyzes all the other EBNF productions skips over them. The real definitions are hard-coded in the DMIS parser.

4. A comma is represented by a production whose name is **c**. This is to avoid having a mix of literal commas and separator commas, which is very hard to read. For example, the definition of the **dmismnStm** given earlier could also be written as follows, but the eye stumbles at the literal commas.

```
dmismnStm = DMISMN , '/' , stringConst , ',' , versionTag,
  [',', conformItemList], # ;
```

5. In the DEBNF representation of DMIS, the end of a line of a DMIS input file is indicated by the **#** character. This is the only DEBNF convention that violates the rules of EBNF. In a DMIS input file, the end of a line is indicated by a carriage return followed by a line feed, i.e. ASCII 13 followed by ASCII 10.

6. DEBNF does not require any specific syntax or naming convention for lists. Lists that are not optional could be represented in at least three ways. It is not obvious from the right side of a production that a list is being defined. Therefore, to make DEBNF files easier to comprehend, the DEBNF files in the test suite use two conventions. First, the name of a list always ends in "List". Second, the definition of a simple list (repetitions of a single type of item) usually has the following form.

```
itemList = [itemList , c] , item ;
```

10.6 What DEBNF files are available?

The ebnf directory contains two versions of the DEBNF file for full DMIS. One has no comments and does not define a CALL block (since there is no such thing in DMIS). The other has a lot of comments and does define a CALL block (since CALL blocks exist in the file prepared by the preprocessor). The DMIS file printer in the dmis.cc file knows that only the first line of a call block should be printed, so the call block disappears (as it should) if a DMIS file is printed back out again.

Annex C of DMIS 5.2 contains a 104-page DEBNF file for all of DMIS. The dmisFull.debnf file in the ebnf directory differs from Annex C of DMIS 5.2 but describes the same syntax.