GFP0805

Generating and Verifying Data for Testing Backup Systems

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

_____

Patrick McAnneny

_____

Dylan Streb

Date: April 30, 2009

Approved:

_____

Professor Gary F. Pollice, Major Advisor

Abstract

Testing backup and recovery software is expensive, requiring the creation and storage of large, complex data sets. Our project reduces this overhead for EMC Corporation. We developed a grammar to define data sets and a Domain Specific Language processor for that grammar. This approach allows for the small definition to persist rather than the actual data set. These definitions are used to generate data sets or to validate recovery.

Acknowledgements

We would like to thank EMC Corporation for sponsoring the project, as well as our contact at EMC, John Rokicki, for providing advice and giving feedback on our project. We would also like to thank Professor Gary Pollice for advising the project.

Table of Contents

List of Figures

Introduction

The main goal of our project was to develop a system for EMC Corporation to reduce the overhead cost of testing backup, recovery, and archiving solutions. To test scenarios in these types of applications, data sets need to be created. These data sets need to be large and complex enough to thoroughly test these systems. Herein lies the problem; creating complexity in a data set requires time, and storing all of these data sets requires storage space.

To solve this problem, our project implemented two major steps. First, we created a Domain Specific Language[1] to describe a data set, so that a small definition file could be stored, rather than the data set itself. Second, we developed a program that can use this definition file to generate the data set it defines, or verify that an existing data set adheres to the description of a definition file.

Our project greatly reduces the amount of storage space necessary to store these data sets. A data set definition file is very small—on the order of kilobytes—whereas a data set itself can be tens of thousands of times larger. Not only does this save lots of storage space, but it also lets developers share data sets or collaborate. Our project also reduces the time needed by developers to create data sets. Instead of taking the time to manually create thousands of files with specific attributes and data, our program can be used to define and automatically generate them.

In the following background section, the reason for our project will be discussed in greater detail, as well as who our sponsor is and what they do. Other aspects of our project will also be described in greater detail, such as data sets. In the methodology section, our development process will be described, and we will discuss the key decisions

we made. Then, in the results and analysis section, we will provide a short analysis about what the actual results of our project were, and how EMC liked the project. Finally, the future work and conclusions section will provide insight as to what the future holds for this project, since another development team could take over and, for example, add graphical elements to it.

1. Background

In fall of 2007, John Rokicki, who would later become our contact at EMC, realized the need for the creation of a software tool that would simplify the testing of data integrity applications. This insight, which became the basis of our project, was inspired by an inefficient use of resources that occurs when attempting to test various scenarios in 'BURA', or backup, recovery, and archiving solutions. The problem is that data sets are needed for these tests, and data set creation is not well controlled or defined, and can take considerable time, depending on how complex or non-trivial the data set needs to be for the given test. Furthermore, these data sets need to be stored, and if each data set is on the scale of gigabytes, this quickly adds up.

A data set can be defined as the layout of data on a disk, including the directory and file structure, content of files, and the metadata of all files and folders; all of the files and folders that compose a person's hard disk can be thought of as a data set. This type of data set would be used for file-system testing. Testing an application, on the other hand, would require the generation of an application specific data set that the application could use. The primary focus of our project is file-system data set generation, but it is possible for the design of our system to be extended to allow for the creation of application specific data sets.

Data sets are needed for testing scenarios in backup, recovery, and archiving applications. The initial data set is used by the application to create, for example, an archive or backup. After this backup or archive is made, the data set can then be re-assembled from it. This newly created data set is then checked for consistency with the original data set. If something does not match up, this process shows that something is

wrong with one of the steps in the archiving or backup process. The storage space required to perform these tests is at least twice the size of the data set being used; the data set is initially stored for use in the test, a backup of it is made during the test, and in the mean time, the developer who originally created the data set probably has a copy of it as well. Performing a verification test inherently requires at least twice the size of the data in storage space, because an exact duplicate is needed to test for consistency during the final step of a test. Accordingly, performing a lot of these tests requires large amounts of storage space.

Also, creating these data sets is not a quick task, requiring developer time to make each data set customized specifically to test a particular aspect of software, or a specific scenario. Furthermore, data set creation is not well defined or reproducible across developers. Since creation of data sets is cumbersome, each developer tends to have his or her favorite data set that becomes adapted for his or her own individual tests. Also, developer collaboration on this subject is minimal, so lots of similar work gets unnecessarily repeated.

Our project aimed to remove these obstacles by creating a Domain Specific Language (DSL)[1] to specify a definition of a data set that can be used to create a data set for use with backup solutions. This definition can be persisted instead of the data set it describes. The size of the definition is negligible when compared to data sets potentially containing gigabytes of data. Using a definition also speeds the development of data sets, as it is only necessary to manually create the definition and let the program generate the files, rather than manually creating, setting the directory structure of, setting metadata for,

and filling the contents for every file within the data set. Furthermore, this design allows for easier collaboration amongst developers.

About Domain Specific Languages

A Domain Specific Language is a high-level programming language that is designed to solve a narrow set of problems. This is in contrast to a general programming language such as C, which is intended to be useful for a wide variety of tasks. A DSL is intended for use on a well defined set of problems and will have little to no use for any other problems. Because a DSL is intended only for a narrow set of problems, it is simpler to use a DSL for solving problems within its scope than a general purpose language; this is the motivation behind creating a DSL.

The tool used to create the DSL is ANTLR, ANother Tool for Language Recognition[2]. ANTLR is a top-down parser generator that is written in Java and can output code to a number of different programming languages. ANTLR is used to interpret a language by giving it productions to follow in the form of a formal grammar. A formal grammar is a set of formation rules that can be used to describe a set of strings within the alphabet that are syntactically valid. The process of taking an input string and testing it against a grammar is known as parsing. Parsing is typically done in 3 main steps. First the input string is analyzed for known substrings, creating a stream of tokens as opposed to a stream of individual characters. This is known as lexical analysis. The stream of tokens is analyzed based upon its order to construct an Intermediate Representation (IR), commonly an Abstract Syntax Tree. This is known as syntactic analysis. The IR is in the form of a common data structure that can be easily interpreted by another program.

Actions can be executed as the nodes of the IR are traversed. This is the basic premise behind an interpreted programming language.

2. Methodology

The first task for any system design is to gather the requirements. We met with John early in the year to discuss what the system had to do and what constraints it had to follow. We briefly discussed the system and requirements and had him send us a more formal document specifying the requirements. We met again a few days later to discuss the project in greater depth, due to problems understanding what the system was supposed to do. He went more in depth on the problem the system was to solve, how it was being handled currently, and how it would be valuable to the company. With the ambiguities surrounding the proposed functionality of the system removed, we began investigating ways to solve the problem.

The main tasks of the project were to create a way to define data sets through some means, and then create a program to construct a defined data set or verify a data set by use of a definition. The requirements document provided to us by our contact at EMC proposed using XML to describe data sets, so we briefly considered parsing an XML file to create a data set. At the next MQP meeting, however, our advisor suggested that we look into using a formal grammar to create a Domain Specific Language for describing data sets rather than using XML.

This was around the time when we developed a basic plan as to how our system would function. Our Domain Specific Language would be used to specify a data set in a small text definition file, which would then be read in by our first program, 'build.' The build program would then construct a data set based on the definition file. Our second program, 'verify,' would take a data set and a definition file, and output as to whether or

not the specified data set conforms to the definition provided. Our system architecture can be seen in the following diagram.

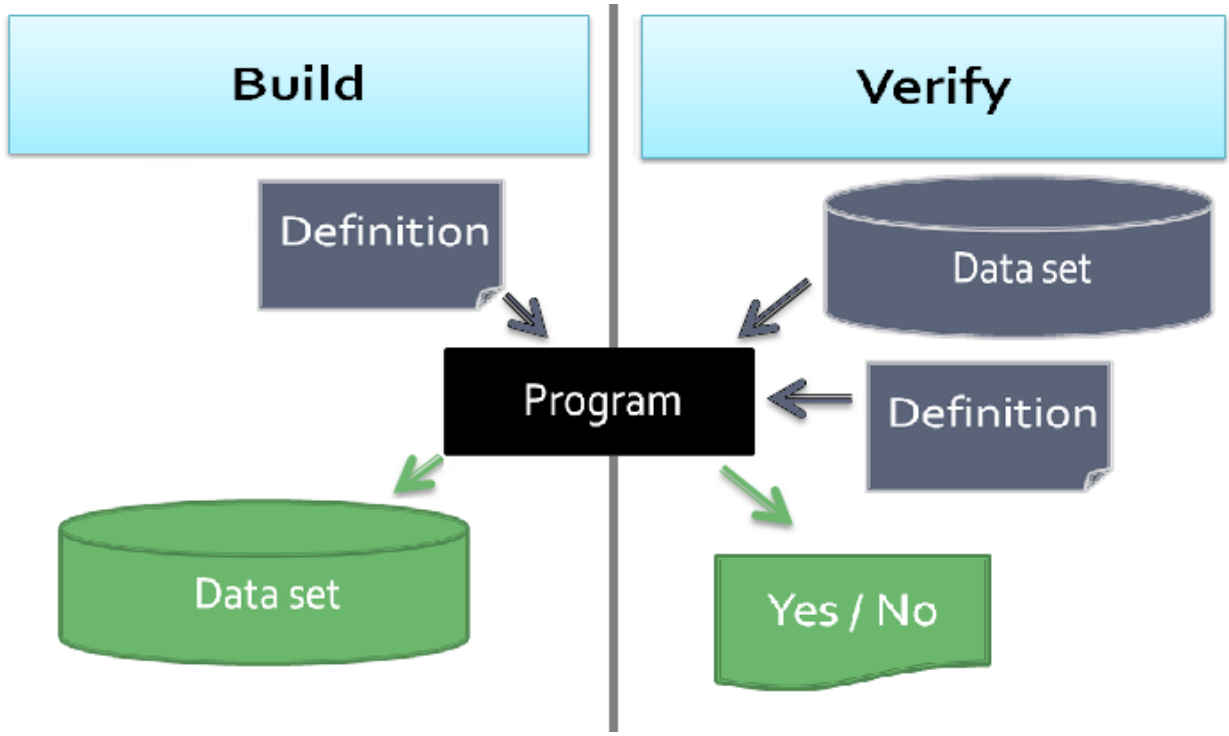

Figure 1

Now that we had our basic system architecture planned out, we started working on our Domain Specific Language. We first drafted a grammar using a set of tools for language processing called lex and yacc, but decided to switch to using ANTLR when it was recommended by our advisor. ANTLR was chosen in favor of yacc/lex because we believed it would be easier to use and could generate object oriented code.[8] One of the requirements was that the code we write must be in C, C++, or Perl. ANTLR is written in Java, but can output code to a variety of languages, including C++. We were not able to generate output code in C or C++ under Windows using ANTLRv3, the latest version,

but were able to use ANTLR-2.7.1 with C++ output under MSYS using MinGW g++ as the C++ compiler.

After getting the tools we needed, we began working on the formal grammar. The requirements included being able to create tens of thousands of files and directory structures a thousand levels deep. It was obviously not feasible to define each file individually, so our idea was to define a small number of file definitions, create as many copies of these definitions as desired, and give each copy a unique and meaningless name to avoid file system conflicts. We decided to divide the grammar into 3 parts; "globals" for certain variables mentioned in the requirements that would affect every file, "definitions" for these file definitions and all the associated metadata, and "root" for a list of files that would be created at the target directory. Directory structure was handled by having a list of files, structured in the same way as the 'root' section, included in the metadata for directories with the limitation that only previously defined files could be included within a directory.

To accommodate for the large number of files needed, simple algebra was added to the grammar. The multiplication operator, '*', followed by the number n means to create n copies of a defined file or directory. The operator '^', only applicable for directories, followed by the number n means to create the directory and all of its contents, then create a copy of the directory inside of it, and a copy of the directory inside of that directory, until the depth of the final directory is n. These two operators would allow the user to specify a massive set of files with minimal effort. This initial design of the grammar did not change during the project.

An example of a valid data set definition file that uses our Domain Specific Language can be seen in Appendix A. This file adheres to the latest rules of our Domain Specific Language, and specifies the file/directory structure as seen in illustration two.

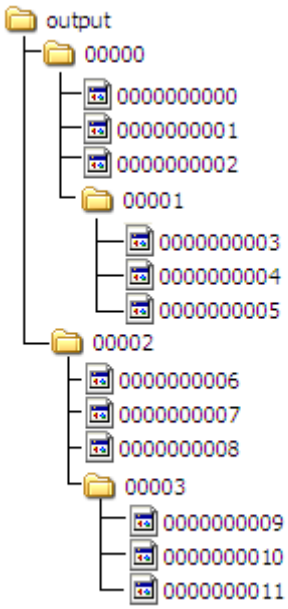

Figure 2

The next step was to create the program that would invoke the functions created by the ANTLR grammar. The requirements document mentions two programs; a program to create a dataset and a program to verify a created dataset. For simplicity, we began working only on the creation program. First, it was necessary to tackle the issue of filenames. Filenames must be unique within a single directory without being individually named by the input file, and the requirements only state that the length of the filename must be specified. The solution was to specify an alphabet that filenames could use, and have filenames be numbered sequentially for a given length. This alphabet can include almost any character and will filter out duplicates. If the operating system is not case

sensitive, the program will need to treat the lower case and capital versions of letters as equivalent as well.

Filenames are determined by treating the strings as a number, with base equal to the number of characters in the alphabet. For example, if the alphabet is “abc”, filenames are treated as base 3. The value of each "number" in the alphabet is equal to its position in the alphabet, with the rightmost character having a value of 0 and the leftmost having a value of base minus one. When the first filename of length x is requested, the string corresponding to the 0 value is returned, e.g. if a filename of length 8 was requested using the previous alphabet the first filename would be "aaaaaaaa". Subsequent requests for a filename of the same length will return the string with a value one greater than the previous; continuing the previous example, the next filename would be "aaaaaaab" and the filename after "aaaaabcc" is "aaaacaa" Creating a file has no effect on the filenames of any other length. If more files of a specific length are specified in the input file than can be created due to name limitations, an error is thrown and the program quits. The maximum number of files of a given length is equal to (alphabet length)^(filename length). We were able to create a working prototype that demonstrated that the input file was being parsed correctly and that file names were being generated as expected.

The next step was to create the files and fill in the contents. We did not know exactly how we wanted to specify the contents of the files. The requirements only stated single byte fill and a repeating range of bytes as necessary, but the original proposal had mentioned a desire for the creation of application-specific data. We wanted a method of specifying file contents that could be extended for future development Simply hard coding the single byte fill and repeated range of bytes into the delivered package would

just cause problems in the future; however without a method to solve this problem we decided to hardcode single byte fill as a placeholder and implement a better solution later.

The next step for creating files was to set the metadata for the files. This was a problem, as the Windows metadata we had to set had no direct equivalent in the C++ standard. Because we were using an emulated Linux environment for development, we had no way to set any of the file attributes, with the exception of read-only. Read-only could be changed with the chmod() function, but there were no other system calls available to change the other attributes. The only solution we were able to find was to use the system() call to run the windows Attrib command, however this is a very slow solution. We were not ever able to find a better solution for Windows, so the poor implementation of Windows' metadata remains in the current deliverable.

With this, files were created with contents and metadata that could then be verified. We moved functions that would be needed for both building and verification into a common source file, so creating the verification program only required modifying a few functions. All of the functionality for parsing the input file, navigating the tree generated by ANTLR, and creating a list of files to act upon is done within the common source file; the source files exclusive to building and verifying only specify how to act upon this list of files. Revisions of the source code were able to move even more functionality into the common source file, until all that remained in the build/verify source files were redirection functions for handling files to specify if files should be created or verified. The actual creation and verification of the files is done in another source file because creating and verifying files requires operating system specific code and so was moved to increase compatibility.

The final significant change to the program was the creation of the filedata class. The filedata class is a layer of encapsulation; its responsibility is generating data that is used to fill or verify files, maintaining an internal state as necessary. A file can have any number of filedata objects associated with it. When the file is created or verified, the file's filedata objects are called cyclically, getting data from each one until the size limit for the file is reached. The encapsulation of this behavior also helps make it extendable for future development. The current implementation only allows for repetition of integer values, strings, ranges of numbers, and a random number generator to create uncompressible files. It is possible to add a new data fill type by making small changes to the grammar file and by updating the filedata class to handle the new type. This can be used to create application-specific data, such as working PNG or Bitmap images, WAV files, etc, as long as the format for the filetype is known; doing this is mentioned more in-depth in the Future Work section.

The last requirement that we fulfilled was creation of symbolic links (symlinks). Symlinks would have to point to a file created in the dataset, so it was necessary to devise a way to represent this with the grammar and update it. The main problem to overcome was that file creation is done with groups of files, while symlinks only point to a single file. Having a link point to a group of files is not possible; it must point to one of the files in the group. The solution we came up with was to have a group of symlinks and have this group point to a group of files (links to directories or to other links are not supported). The links within the link group will point to the files within the file group sequentially; the first link points to the first file, the second link points to the second file, etc. If there are more links than files, the links will wrap around to the beginning. For

example, if there are 20 links and only 2 files, each file will have 10 links pointing to it. The groups of files are specified by an operator in the grammar, while the groups of links are simply every link of the same definition. Groups of files are specified by adding an arrow (->) and a link variable at the end of a file variable when a file is specified. For example, (F1*10->S1, F2*2->S1, S1*20) will create 20 links of type S1 and 12 files that the links of type S1 will point to. Symlinks are not present in Windows and have no emulation in MSys, so symlinks only have an effect on Linux. Trying to specify the creation of a link in Windows will not create any additional files, nor throw an error.

3. Results and Analysis

The result of this project is a working system. The system meets many, but not all, of the requirements that were given to us. Using the Build program it is possible to create large numbers of files and to specify patterns for the filenames, directory structure, contents, and metadata. The files are created and can be checked by the Verify program which can detect if any of the contents for any of the specified files have changed, if there are any files or directories missing (or renamed), and if any unexpected files were found in the target directory. Thus the system meets the primary requirement. However the system is far from complete or optimal, and needs additional work.

Several features are missing from the deliverable. The filedata class is currently only capable of handling strings and "NOISE". Linux support is largely untested, and verifying symlinks is missing completely. Setting metadata in Windows is very inefficient with the current method, and verifying metadata is missing. Unicode support for filenames is not present due to incompatibilities with MinGW, our compiler.

File creation and data filling is very inefficient and needs optimization. File creation is slow due to the problem of setting metadata in Windows by using a system() call. This greatly slows down file creation, but we were not able to find a solution that worked in MinGW. The only optimization we were able to add are that the system() call is not used if none of the file attributes are set to true; they all default to false on file creation, so setting them in this situation is unnecessary. The speed difference in setting metadata is very clear. Creating 10,000 files with no contents without setting the metadata took 10 seconds. Creating 10,000 equivalent files with setting metadata took 15 minutes, 59 seconds. This shows that setting a file's attributes takes nearly 100 times as

long as creating the file itself. The problem is exacerbated because the files have no content, but is always present when file attributes are set – plus it must be remembered that directories have attributes but cannot have contents themselves. Creating 2048 files of 1 megabyte each (total size of 2 gigabytes. File contents used was a simple "a") took 20 minutes, 17 seconds. The same procedure with setting file attributes took 23 minutes, 18 seconds.

Filling the contents of files is also slow due to inefficiencies in the filedata class and in the function that calls the getData() function. Creating a single 2 gigabyte file with the pattern "a" took 20 minutes, 6 seconds. Creating a single 2 gigabyte file with the pattern "aaaa" took 7 minutes, 2 seconds. The reason using a longer pattern is faster is because getData() is called one fourth as many times in the latter example. Larger patterns further reduce the execution time. Using a single string of 8 characters took 4 minutes, 53 seconds. Using a string of 64 characters took 3 minutes, 19 seconds. The main inefficiencies appear to be the ANTLR runtime tree navigation functions (getNextSibling() and getFirstChild()) and dynamic allocation of memory returned by the getData() function. In the current design, getNextSibling() is used to cycle to the next node containing file data, and getFirstChild() is used to return to the first node when the final node is reached. These functions are slow because they must account for runtime changes to the tree. The structure of the tree does not change after the initialization of the filedata class, so it is possible to retrieve the necessary nodes once and store pointers to them in a less flexible and therefore faster data structure. Using the pattern "a" "a" instead of just "a" will halve the number of calls to getFirstChild() while changing nothing else. This results in a small speed improvement; 19 minutes, 50 seconds

compared to 20 minutes, 6 seconds for “a”. The other improvement that could be made is to use static memory instead of dynamic memory allocation. To ease compatibility between the node types, getData() allocates memory for each block of data it returns, which is then freed by the calling function. A method of returning data that used static memory while still accommodating the potentially variable lengths of the nodes would be faster and would be a better design.

Statistics:

Grammar file is 461 lines. Lexer has 51 rules. Parser has 38 rules.

Summing the length of every source file except OSspec_linux and the ANTLR generated files is 2447 total lines of code.

There are a total of 69 functions and 7 classes.

4. Future Work and Conclusions

Software Bugs

Due to limited development time, our system has a number of missing features and software bugs. This section lists all of the known deficiencies in the system. Many are simple to solve, but were not discovered before the code freeze.

Setting metadata in Windows currently uses a very inefficient placeholder method, as we were not able to find a system call to change a file's attributes using MSYS. Setting metadata in Windows currently uses the system() function to run the program attrib. This is a very inefficient way of setting metadata that causes slowdowns and can cause errors when creating lots of files.

Additionally, the verification program does not check the metadata for files. While it is possible to use system() to verify the metadata, the code required to do so would be complicated and in no way portable to a proper implementation of metadata handling. We decided to not verify metadata.

Symbolic links in Linux are not verified. Exposure to an actual Linux system during development was minimal. The system calls required for checking a symlink were not known when we tested the system on a Linux machine, so the function was never written.

Another result of the limited exposure to Linux during development is that verifying permissions in Linux is not tested. The function does exists and should work, according to the documentation for stat, but this cannot be confirmed.

Our system has no support for Unicode filenames, which is one of the requirements. MinGW does not support some features that would be required for proper Unicode support, including most wstring functions and different locales. A workaround for this problem was attempted, but abandoned.

Error handling when interpreting the input file is extremely lacking. Errors from the ANTLR runtime attempting to parse an invalid grammar file are very vague, typically do not give context, and do not give suggestions on what the exact problem is or how it could be fixed.[8] We did not have time to try to have the grammar give better error messages. Semantic errors, which are handled by the program and not by the ANTLR runtime, are typically better.

It is possible to specify files without read permission in Linux. The verification will not be able to succeed with these files because it cannot read them. It is necessary to change the file's permission to allow reading, verify the contents of the file, and then reset the permissions back to the original state to avoid potential verification issues with a second execution of the application. There is no way to disable read access in Windows using the build program, so this bug is only present in Linux.

Due to a bug, the maximum filesize for a single file is limited to 2 gigabytes. Internally, numbers are represented using the size_t type, limiting all integers to a maximum value of 4 gigabytes. It is necessary to change this to 64 bit unsigned integers to enable larger file sizes. Additionally, a bug in the conversion of numbers from strings additionally limits the maximum to 2 gigabytes, as the function strtol returns a signed long, which has a maximum value of 2,147,483,647. It should be noted that negative numbers are not required by the program anywhere, and are not supported by the

grammar. The conversion from string to integer returning a signed integer is the result of an oversight.

Future Work

The creation of application specific data was mentioned in the original proposal, but was considered too advanced to be put on the requirements document, so application specific data is not part of the deliverable. Implementing application specific data would likely require a definition for a set of files with a specific layout and naming scheme and special contents of files. Application specific data spanning multiple files will likely need some way of referencing the names of other files within the file contents. This is not possible in the current implementation, as file and directory names are randomly generated and are not accessible in any other part of the program; e.g. the third file created cannot 'see' the names of the first two files, or even be aware of their existence. Implementing such a feature will require a major design change to how filenames are created and how file data is generated. Changing the contents of files should be easier. The filedata class encapsulates the generation of file contents, and the build/verify functions call getData() until they have reached the file's specified size. It is therefore possible to add functionality to the filedata class to generate more specific types of data and to the grammar to specify this data. A simple example would be creating a Bitmap image. It is simple to update the grammar to specify a file's contents as an image of a specific height and width and to then have the filedata class create the appropriate header information and height*width randomly generated pixels (by using the included version of the Mersenne Twister random number generator) and returning this information. However, in the current design, it is necessary to return the entire image at once, which is not desirable from an efficiency or design standpoint.

Another potential topic of future work for our project is to further reduce the amount of human work by allowing more general specifications of data sets. Our method of solving the original problem was to allow a user to specify a large number of copies of a single type of file or directory, saving the user time by removing the necessity of specifying each file individually. A further improvement would be to allow a user to specify a rough description of the entire data set, and use this description to create a dataset definition file that could be interpreted by our system. This would save user time when creating complex data sets by only requiring an overview of the desired data set, allowing the interpreter to create the complexity of the data set instead of the user. Because this rough description would create a reproducible definition, the description itself does not have to be reproducible; e.g. it would be possible to specify the number of files as between 3000 and 5000 and use a random number generator to determine the exact number of files and even how they are placed within the directory structure. Interpreting this rough description would create a reproducible data set definition, so it is not necessary that a second interpretation of the same rough description create the same data set definition. This is beneficial, as a single description can then create multiple data sets if it is necessary to test a system with more than a single data set.

Conclusions

The final result of our project is a working system. The system meets many, but not all, of the requirements given to us by EMC. The system is able to interpret a dataset definition file that we defined and can create the described files with a directory structure, set the contents, and verify the file structure and file contents. The verification can detect changes made to the contents of the files, file deletion, and additional files being added to the target directory. We were not able to implement proper verification of file metadata or symlinks in Linux, however the design is fully done for these, and implementing them should be simple. The file content generation is currently very limited, but should be extendable to allow more complex data fill with minor changes to the grammar. Additionally, the grammar supports some data fill types that were not implemented in the filedata class; finishing the implementation of those should be trivial. The largest deficiency is the lack of Unicode filename support. We were not able to implement this due to an incompatibility with MinGW. A small amount of commented out code exists that may be relevant to implementing Unicode support, however it is completely untested and not close to the full design change necessary to implement Unicode.

Our system significantly reduces the amount of storage space required to store data sets. A data set definition file is very small when compared to a data set itself, which can be thousands or millions of times larger. This saves storage space, especially when it is considered that multiple copies must exist for any data set used for backup. The reduction in size also eases sharing of data sets among developers, as does the relative rigidity of using an automatically generated data set over a manually generated data set.

Development time is also reduced. Creating files without needing to wait for additional user input is faster. Not requiring user input also means that the user can set the machine to create a dataset while the user can perform other tasks. If the user instead must interact with the machine to create the data set, data set creation involves machine time and user time. It is necessary for the user to create the definition file, but this can be done in a standard text editor and is not load intensive, and the machine can perform other tasks for other users during this time. Finally, the algebraic expressions for file creation and file patterns for file contents mean that large data sets can be specified by a user faster using our system than by hand.

In short, our system can reduce the cost of testing backup and storage solutions by reducing the storage space necessary to store data sets for these tests, and by reducing the time developers need to spend creating data sets for these tests.

Appendix A Sample input file

```
BEGIN
GLOBAL (
     TARGET = "c:\\output"
     ALPHABET = "0123456789"
     )

DEFINITIONS
FILE F1   (
          NAMELEN=10 SIZE=1000 DATA=NOISE(1);
          ARCHIVE = FALSE
          READONLY= FALSE
          HIDDEN =  FALSE
          SYSTEM =  FALSE
          )

DIR D1    (
          NAMELEN = 5
          ARCHIVE =  FALSE
          READONLY = FALSE
          HIDDEN =   FALSE
          SYSTEM =   FALSE
          CONTENTS = (F1*3)
          )

ROOT       (D1^2*2)
```

Appendix B Grammar File

```
header {
//This appears in every .hpp file:

#include <string>
#include <sstream>
#include <iostream>
#include "common.hpp"

using namespace std;


//Needed for assignments in rule [f/d/l]def. ANTLR throws syntax errors when trying to
put -> into the Tree Walker value assignments.
#define ZNAMESIZE z->namesize
#define ZFILESIZE z->filesize
#define ZPATTERN z->pattern
#define ZARCHIVE z->flags.archive
#define ZREADONLY z->flags.readonly
#define ZHIDDEN z->flags.hidden
#define ZSYSTEM z->flags.system
#define ZPERMISS z->flags.permissions

#define XNAMESIZE x->namesize
#define XARCHIVE x->flags.archive
#define XREADONLY x->flags.readonly
#define XHIDDEN x->flags.hidden
#define XSYSTEM x->flags.system
#define XPERMISS x->flags.permissions

#define WNAMESIZE w->namesize
#define WARCHIVE w->flags.archive
#define WREADONLY w->flags.readonly
#define WHIDDEN w->flags.hidden
#define WSYSTEM w->flags.system
#define WPERMISS w->flags.permissions

}

options {
        language="Cpp";
}


class FileLexer extends Lexer;
options {
        k = 4;
        defaultErrorHandler=false;
        //charVocabulary = '\40'..'\176';
        charVocabulary = '\0'..'\377';

}

COMMENT :       '#' (~('\n' | '\r'))*  {_ttype = ANTLR_USE_NAMESPACE(antlr)Token::SKIP;} ;
GLOBAL  :        "GLOBAL";
BEGIN   :       "BEGIN";
DEFINITIONS:    "DEFINITIONS";
TRUE    :        "TRUE";
FALSE   :       "FALSE";
EQ      :       '=';
COMMA   :       ',';
MULT    :       '*';
EXP     :       '^';
FILE    :       "FILE";
DIR     :       "DIR";
LINK    :       "LINK";
```

```
INT     :       ('1'..'9')('0'..'9')*;
HEX     :       "0x" ('0'..'9' | 'A'..'F')*;
OCT     :       '0' ('0'..'7')*;              //'0' is now valid and will be registered as
an octal integer
FILEVAR :       'F' ('0'..'9' | 'a'..'z')*;
DIRVAR  :       'D' ('0'..'9' | 'a'..'z')*;
ROOT    :       "ROOT";
NAMELEN :       "NAMELEN";
SIZE    :       "SIZE";
DATA    :       "DATA";
UNICODE :       "UNICODE";
SYMLINKS:       "SYMLINKS";
TARGET  :       "TARGET";
//TODO FILEPATH is just a string
//FILEPATH:     '['! ('0'..'9' | 'a'..'z' | 'A'..'Z' | '.' | ':' | '/' | '\\' | '\'' | ';'
| ',' | '-' | '=' | '+' | '_' | ')' | '(' | ' ' | '@' | '#' | '~')* ']'!;
ARCHIVE :       "ARCHIVE";
READONLY:       "READONLY";
HIDDEN  :       "HIDDEN";
SYSTEM  :       "SYSTEM";
CONTENTS:       "CONTENTS";
OPAREN  :       '(';
CPAREN  :       ')';
WS      :       (' '|'\t')+ {_ttype = ANTLR_USE_NAMESPACE(antlr)Token::SKIP;} ;
NL      :       (('\n' '\r') | '\n'|'\r') {newline(); _ttype =
ANTLR_USE_NAMESPACE(antlr)Token::SKIP;} ;
NOISE   :       "NOISE";
OBRACK  :       '[';
CBRACK  :       ']';
DOTDOT  :       "..";
QUOT    :       '"';
SEMICOLON:      ';';
PREFIX  :       "PREFIX";
SUFFIX  :       "SUFFIX";
ALPHABET:       "ALPHABET";
MAXSIZE :       "MAXSIZE";
MAXPATH :       "MAXPATH";
STRING  :       '"'! (CH)* '"'!;
ARROW   :       '-' '>';
CHAR    :       '\''!   (
                        CH
                        )
                '\''!;
LINKVAR :       'S' ('0'..'9' | 'a'..'z')*;

protected
CH      :       ~('"' | '\'' | '\\')
        |       '\\'
                        (       'n'     {$setText("\n");}
                        |       'r'     {$setText("\r");}
                        |       't'     {$setText("\t");}
                        |       '\\'    {$setText("\\");}
                        |       '\''    {$setText("\'");}
                        |       '\"'    {$setText("\"");}
// HEX
// OCT
                        )
        ;
PERMISSIONS:    "PERMISSIONS";

class FileParser extends Parser;
options {
        buildAST = true;
        k = 4;
        defaultErrorHandler=true;             //Turn back to false to manually catch parse
exceptions.
}

start   :       BEGIN^ global defines root
        ;
```

```
boolean :       (TRUE | FALSE)
        ;

number  :       INT
        |       HEX
        |       OCT
        |       CHAR
        ;

global  :       GLOBAL^ OPAREN! (gspec)* CPAREN!
        ;

gspec   :       unicode
        |       symlinks
        |       target
        |       prefix
        |       suffix
        |       alphabet
        |       maxsize
        |       maxpath
        ;

unicode :       UNICODE^ EQ! boolean
        ;

symlinks:       SYMLINKS^ EQ! boolean
        ;

target  :       TARGET^ EQ! STRING
        ;

prefix  :       PREFIX^ EQ! STRING
        ;

suffix  :       SUFFIX^ EQ! STRING
        ;

alphabet:       ALPHABET^ EQ! STRING
        ;

maxsize :       MAXSIZE^ EQ! number
        ;

maxpath :       MAXPATH^ EQ! number
        ;

defines :       DEFINITIONS^ (define)*
        ;

define  :       filedefine
        |       dirdefine
        |       linkdefine
        ;

filedefine
        :       FILE! FILEVAR^ OPAREN! (fdspec)* CPAREN!
        ;

fdspec  :       namelen
        |       size
        |       data
        |       archive
        |       readonly
        |       hidden
        |       system
        |       permissions
        ;

permissions :   PERMISSIONS^ EQ! number
        ;
```

```
namelen :       NAMELEN^ EQ! number
        ;

size    :       SIZE^ EQ! number
        ;

data    :       DATA! EQ! NOISE SEMICOLON!
        |       DATA! EQ! NOISE^ OPAREN! number CPAREN! SEMICOLON!
        |       DATA^ EQ! datav SEMICOLON!
        ;

datav   :       //Blank
        |       number datav
        |       STRING datav
        |       range datav
        ;

range   :       OBRACK! number DOTDOT^ number CBRACK!
        ;

archive :       ARCHIVE^ EQ! boolean
        ;

readonly:       READONLY^ EQ! boolean
        ;

hidden  :       HIDDEN^ EQ! boolean
        ;

system  :       SYSTEM^ EQ! boolean
        ;

dirdefine
        :       DIR! DIRVAR^ OPAREN! (ddspec)* CPAREN!
        ;

ddspec  :       namelen
        |       size
        |       archive
        |       readonly
        |       hidden
        |       system
        |       contents
        |       permissions
        ;

linkdefine
        :       LINK! LINKVAR^ OPAREN! (ldspec)* CPAREN!
        ;

ldspec  :       namelen
        |       size
        |       archive
        |       readonly
        |       hidden
        |       system
        |       permissions
        ;

contents:       CONTENTS^ EQ! OPAREN! fslist CPAREN!
        ;

fslist  :       fsexp (COMMA! fsexp)*
        ;


//An expression may be surrounded by a single parenthesis group for
clarification/emphasis. There is no variability to the order of operations to deal with,
so parenthesis are ignored.
//For simplicity, F * 3 is legal, 3 * F is not.
fsexp   :       FILEVAR^ (exp)* (link)?
```

```
        |       DIRVAR^ (dexp)* (link)?
        |       LINKVAR^ (exp)*
        |       OPAREN! FILEVAR^ (exp)* CPAREN! (link)?
        |       OPAREN! DIRVAR^ (dexp)* CPAREN! (link)?
        |       OPAREN! LINKVAR^ (exp)* CPAREN!
        ;


link    :       ARROW^ LINKVAR
        ;

exp
        :       MULT^ number
        ;

//The order of the * and ^ is not important. If both are present, the order is always
(D^n)*m; or create m directories of depth n.
//Additionally, (D1^n)^m is equivalent to D1^(n+m).
dexp    :       MULT^ number
        |       EXP^ number
        ;

root    :       ROOT^ OPAREN! fslist CPAREN!
        ;

class FileTreeWalker extends TreeParser;
options {
        k = 1;
        defaultErrorHandler=false;
        }

start
{
}
        :       #(BEGIN glob def rt)
        ;

boolean returns [bool t]
{
}
        :       i:TRUE          {t=1;}
        |       k:FALSE         {t=0;}
        ;

number returns [size_t t]
{
size_t a, b;
}
        :       i:INT           {t = strtol( (i->getText().c_str()), (char **) NULL, 10);}
        |       k:HEX           {t = strtol( (k->getText().c_str()), (char **) NULL, 16);}
        |       j:OCT           {t = strtol( (j->getText().c_str()), (char **) NULL, 8);}
        |       l:CHAR          {t = (size_t) l->getText().c_str()[0];}
        |       #(MULT a=number b=number)     {t = a * b;}
        ;

glob
{
bool u=0;       //prevent a warning
dirpath *z;
string s;
}
        :       #(GLOBAL (u=uni | u=sym | z=target | s=prefix | s=suffix)*)
        ;

uni returns [bool i]
{
}
        :       #(UNICODE i=boolean )
        ;

sym returns [bool i]
```

```
{
}
        :       #(SYMLINKS i=boolean)
        ;

prefix returns [string s]
{
}
        :       #(PREFIX s=astring)
        ;

suffix returns [string s]
{
}
        :       #(SUFFIX s=astring)
        ;

target returns [dirpath *z]
{
string s;
}
        :       #(TARGET s=astring)           {z = new dirpath(); z->setPath(s.c_str());}
        ;

astring returns [string s]

        :       k:STRING       {s = k->getText();}
        ;

def
{
filespec *f;
dirspec *d;    //to stop a warning
linkspec *l;
}
        :       #(DEFINITIONS (f=fdef | d=ddef | l=ldef)*)
        ;

fdef returns [filespec *z]
{
z = new filespec();
}
        :       #(FILEVAR (ZNAMESIZE=namelen | ZFILESIZE=size | ZPATTERN=data |
ZARCHIVE=archive | ZREADONLY=readonly | ZHIDDEN=hidden | ZSYSTEM=system |
ZPERMISS=permission)*)
        ;

ddef returns [dirspec *x]
{
x = new dirspec();
}
        :       #(DIRVAR (XNAMESIZE=namelen | contents | XARCHIVE=archive |
XREADONLY=readonly | XHIDDEN=hidden | XSYSTEM=system | XPERMISS=permission)*)
        ;

ldef returns [linkspec *w]
{
w = new linkspec();
}
        :       #(LINKVAR (WNAMESIZE=namelen | WARCHIVE=archive | WREADONLY=readonly |
WHIDDEN=hidden | WSYSTEM=system | WPERMISS=permission)*)
        ;

namelen returns [size_t a]
        :       #(NAMELEN a=number)
        ;

size returns [size_t a]
        :       #(SIZE a=number)
        ;
```

```
data returns [filedata *d]
        :       i:NOISE                         {d = new filedata(i);}
        |       k:DATA                          {d = new filedata(k);}
        ;


range
{
size_t s;
}
        :       #(DOTDOT s=number s=number)
        ;

archive returns [bool b]
        :       #(ARCHIVE (b=boolean) )
        ;

readonly returns [bool b]
        :       #(READONLY b=boolean)
        ;

hidden returns [bool b]
        :       #(HIDDEN b=boolean)
        ;

system returns [bool b]
        :       #(SYSTEM b=boolean)
        ;

contents //returns...
        :       #(CONTENTS (FILEVAR | DIRVAR) )
        ;


permission returns [int i]
        :       #(PERMISSIONS i=number)
        ;


rt
{
}
        :       #(ROOT (FILEVAR | DIRVAR))
        ;
```

Appendix C Program Pseudo-code

```
Main
        Parse input file //ANTLR Runtime. Generates the tree
        get Globals, Definitions, and Root tree nodes
        get arguments from Globals
        HandleDefinitions()
        foreach filevar V in Root
                CreateFile(V)


HandleDefinitions
        foreach definition A in treenode
                assert A's name not already used
                if A is a Directory and A's Contents is not empty
                        foreach definition B in Contents
                                assert B's name has been used
                insert A into list of filedefs or dirdefs

CreateFile on filevar V
        assert V's name is valid
        get V's multiplicity as M
        if V is a directory     then get V's depth as D
        for i from 1 to k
                getfilename()
                if V is a file  then HandleFile()
                if V is a directory     then HandleDirectory()
                if V is a directory and V's Contents is not empty
                        foreach filevar c in Contents
                                CreateFile(c)
                        store Current Working Directory as cwd
                        while (D > 1)
                                CD into most recently created directory
                                CreateFile(copy of V)  //A single directory
                                D--
                        CD to cwd

HandleFile
        if program is Build
                create file
                set file's metadata as specified by definition
                while file's size is less than definition size
                        Append getData() to file
        if program is Verify
                assert file exists
                assert file's metadata matches definition's metadata
                while file position is less than definition size
                        call getData() to get n bytes
                        assert next n bytes from file match return from getData() exactly
                        advance file's position by n

HandleDir
        if program is Build
                create directory
                set directory's metadata as specified by definition
        if program is Verify
                assert directory exists
                assert directory's metadata matches definition's metadata

getData
        Convert the currentdata to a string
        set file's currentdata to the next data in the list
        if end of list was reached      then return to beginning of list
        return the data string
```

Glossary

Domain Specific Language - A computer language that is targeted to a particular kind of problem, rather than a general purpose language that is aimed at any kind of software problem.[1]

ANTLR - ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages.[2]

MSys - Minimal SYStem, a Bourne Shell command line interpreter system for Windows providing an alternative to the native cmd.exe.[3]

MinGW- Minimalist GNU for Windows, a port of the GNU Compiler Collection (GCC), and GNU Binutils, for use in the development of native Microsoft Windows applications.[4]

Symbolic Link (Symlink) - A special type of Unix file which refers to another file by its pathname. Most operations upon a symlink, such as reading and writing, will occur upon the referenced file instead.[5]

Hard Link - One of several directory entries which refer to the same Unix file. Hard links to the same file are indistinguishable from each other except that they have different pathnames. They all refer to the same inode and the inode contains all the information about a file.[6]

Unicode - The standard for digital representation of the characters used in writing all of the world's languages. Unicode provides a uniform means for storing, searching, and interchanging text in any language.[7]

Metadata – Data about data. Data that describes a file while being separate from the contents of the file, including the file name, size, and access permissions.

Data set – The physical data (contents), metadata, and directory structure of a set of files.

References

[1] 'DomainSpecificLanguage' Martin Fowler's Bliki

http://www.martinfowler.com/bliki/DomainSpecificLanguage.html (April 21st, 2009)

[2] 'About the ANTLR Parser Generator' ANTLR Parser Generator

http://antlr.org/about.html (April 19th, 2009)

[3] 'MinGW' Minimalist GNU for Windows http://www.mingw.org (April 20th, 2009)

[4] Ibid

[5] 'symbolic link' definition from dictionary.die.net October 10th, 2003

http://dictionary.die.net/symbolic%20link (April 19th, 2009)

[6] 'hard link' definition from dictionary.die.net October 10th, 2003

http://dictionary.die.net/hard%20link (April 19th, 2009)

[7] 'Glossary' The Unicode Consortium February 4th, 2009

http://www.unicode.org/glossary/ (April 22, 2009)

[8] Parr, Terrence *The Definitive ANTLR Reference: Building Domain-Specific Languages*. illustrated edition. North Carolina: Pragmatic Bookshelf, 2007.