Cmpt 473 Assignment 2

Base specification:

xmlutils -> xml2csv : https://github.com/knadh/xmlutils.py

Xml2csv Inputs:

- Input file: XML Document filename
- Output File: The filename of the CSV file we are creating
- Additional arguments:
    - Tag: a node that represents a single record
    - Delimiter: Specify the delimiter for separating items in a row in the CSV file, default is “, “
    - Ignore: A list of element tags in the XML document to ignore
    - Noheader: To exclude the CSV header / first line, default is yes=1, no = 0
    - Encoding: Specify the character encoding of the document, default = utf-8
    - Limit: Specify the number of records to be processed from the document, default = no limit (-1)
    - Buffer: Specify the number of record to be kept in memory before the records are writing to the output CSV file, default = 1000

Xml2csv Outputs:

1. For valid inputs, The CSV file
2. For invalid inputs, error message

For specification of the file formats used in the program tested, we followed these specifications and guidelines:

XML: https://tools.ietf.org/html/rfc3470
CSV: https://tools.ietf.org/html/rfc4180

Input Space Partitioning / Modeling:

The component we are testing is an open source XML to CSV converter, xmlutils: xml2csv. All of the inputs are:

- Input filename
- Output filename
- Tag
- Delimiter:
- Ignore
- Noheader
- Encoding
- Limit
- Buffer

The partitioning characteristics for most of the inputs could be partitioned using boolean (true or false) values. For the few that could not (Some constraints to check size of Limit, Buffer) we used an integer format to represent specific values.

Model:

	Input	Constraints	Values
1	IsInputArgumentProvided		true/false
2	DoesInputFileExist	1 = True	true/false
3	IsInputFileValidXML	2 = True	true/false
4	IsOutputArgumentProvided		true/false
5	DoesOutputFileExist	5 = True	true/false
6	IsTagArgumentProvided		true/false
7	DoesTagExist	6 = True	true/false
8	IsDelimiterArgumentProvided		true/false
9	IsLimitArgumentProvided		true/false
10	LimitSize	9 = True	Default, less than, greater than, equal
11	IsIgnoreArgumentProvided		true/false
12	DoesIgnoreTagExist	11 = True	true/false
13	IsBufferArgumentProvided		true/false
14	BufferSize	13 = True	Default, less than, greater than, equal
15	IsEncodingArgumentProvided		true/false
16	IsEncodingValid	15 = True	true/false

As above, the constraints we applied were essentially necessary to test the validity of a given use. For example if an input filename is provided, we need to check if it actually exists, and then if it exists we need to check and be sure that it is in fact valid XML, otherwise the call to the program would not work.

Combinatorial planning and test generation:

As prescribed, we used the ACTS tool (GUI) to create a test system with our parameters along with their respective constraints (if any). The ACTS tool was then used to generate our pairwise tests. The input combinations of the ACTS were converted into the equivalent calls to the program with relevant arguments. For example, if the IsInputArgumentProvided parameter was true, then the --input argument would be applied to the call of the program. If the IsInputFileValidXML parameter was true it the path to a valid XML file would be used, if it was set to false, then a path to an invalid XML file would be used.

Report and Analysis:
We generated 21 tests, 15 tests were successful / passing tests. If we did not use pairwise testing we would have needed to generate 5832 tests. There were several trade offs we made as a result of using pairwise testing. The primary issue with the pairwise testing is that only 1 of the 21 tests that were generated provided a sequence of inputs that led to the program functioning without error (expected or unexpected). Having overall less coverage and being prone to missing some system bugs is a major trade off in order to have the low cost of pairwise testing implemented. In addition, simple random testing seems to be as effective as pairwise testing. Obviously we did not test every combination of inputs, thus making it clear that coverage is not 100%, and thus susceptible to failure, even though it is widely considered to be good enough.

The assignment showed us everything needed to fully choose and test a system or program. An all encompassing assignment. The simpler parts of the assignment, specification and partitioning actually turned out to be fairly involved and difficult at times. Having to dig deeper into the setup and specifications of rather simple file formats was eye opening. Creating the input space model initially was straightforward, but after going over different test cases and requiring more inputs to partition, the list of requirements grew extensively. We believe that having more direction, and clarifying some of the ambiguity would have made our experience better as initially there were a lot of questions we had. The hardest part of the assignment was thinking of input space partitioning in terms of the formal specifications of the input and output formats as this was so dramatically different from examples we went over during class.