Project Report

for

Automatic Random Regression Testing with Human Oracle

Prepared by: Ivan Maguidhir, C00002614, April 2011

Table of Contents

Introduction

The purpose of this document is to provide an overview of:

- Whether or not the finished product matches the original specification and design
- Problems encountered and their solutions
- New information learned
- Features that I would have liked to include

Description of Submitted Project

The submitted project consists of:

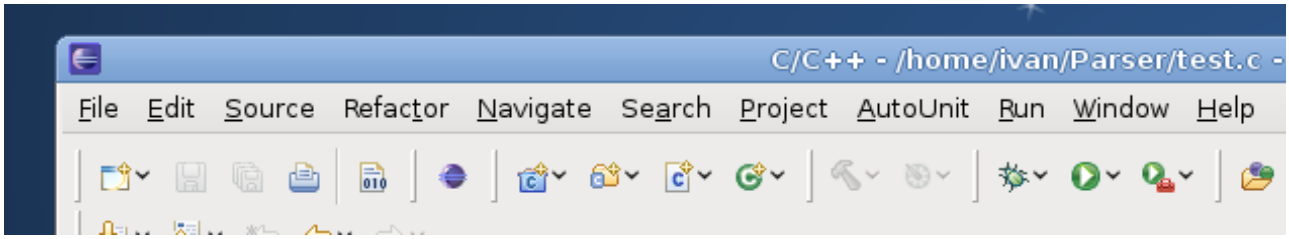
- A library containing the core functionality of the software
- A command-line application

```
user@machine:~$ autounit --help
Usage: autounit [OPTION...] [FILE...]

Examples:
        autounit -cf gcc /usr/bin/gcc   # set the compiler path to /usr/bin/gcc
        autounit --create functions.c   # interactively create tests for functions.c
        autounit --list                 # list all functions under test
        autounit -l calc.c              # list functions under test whose source-code is calc.c
        autounit -r                     # run tests (generally called by a job/task scheduler)
        autounit --remove console.c     # remove all tests associated with the console.c file

        Main operation mode:
        -h, --help                      # display this help screen
        -c, --create file               # create tests for the specified source-code file
        -r, --run                       # run tests
        -l, --list                      # list functions under test
        -rm, --remove file              # remove tests associated with the specified file name
        -cf, --config option value      # set the specified configuration option
```

- A plug-in for Eclipse CDT which adds an additional menu to the IDE



Description of Conformance to Specification and Design

Eclipse Plug-in

I decided along with my project tutor to implement a plug-in for Eclipse. This work was additional to the original specification. The reason for developing the plug-in was so that a developer using

Eclipse could generate tests from within her development environment. The plug-in provides a GUI written in Java which uses the Standard Widget Toolkit (SWT).

Interface

In order to implement the Eclipse plug-in it was necessary to add a Java Native Interface to the library component of the software so that the Java code running inside Eclipse could access the required native C++ functions.

Compatibility

I wanted to write the project so that it would build on multiple architectures. Due to the constraint of the project deadline and on the advice of my project tutor I decided to focus on getting the application working on Linux.

Windows

The problems which need to be solved to build the application on Windows are:

- dlopen API for loading libraries dynamically does not exist. Calls to the dlopen API would need to substituted with the Windows equivalent (e.g. LoadLibrary()).
- The software is currently written for use with GCC, users would need to have this installed or the commands the software uses for compiling (in the Config class) would need to be updated for use with a different compiler.
- The Apache Xerces-C library source-code would need to be added to the project and built with it. On Linux a developer can install Xerces by building it themselves or downloading it pre-built from a repository and it is installed at a standard location on disk.

Localization

The software does not fulfil the requirement of having translatable strings separated from code and in an external XML file. Again this was due to time constraints as I wanted to focus on the main functionality of the software. I really only included it in the Software Requirements Specification to demonstrate I was aware it.

Learning

Technical

Lex & Yacc

I learned how to use Lex & Yacc properly to generate a parser. At the start of the year I was under the impression that Lex & Yacc were used only once to generate code for a parser and that the generated code was then modified until it performed the tasks required. I understand now that the lexicon and grammar are the only things modified and that for each modification Lex & Yacc are called again to generate new parser code. Additionally, I learned how to create parse trees and a symbol table with Lex & Yacc which allow a developer to keep a record of terminal and non-terminal symbols of interest after reductions have taken place.

CppUnit

I studied the history of xUnit framework and became familiar with the CppUnit implementation of it for C++. As the implementations of xUnit for different languages are very similar I shouldn't have much difficulty in writing tests for other languages that I'm familiar with or new languages that I learn in the future. At the start of the year I had no experience of unit testing whatsoever.

C and C++

My knowledge of C and C++ increased during the year:

- Pointer to an array vs. Array of pointers
  A pointer to an array is not the same thing as an array of pointers. Common-sense would indicate that these are not the same, however I thought in C and C++ that they were the same variable type. I'm sure I have encountered problems in the past as a result of thinking this.
- A pointer points to a variable not a value
  A few times while I was working on the code-generation part of the project I tried to use the logical AND operator “&&” on a variable to reference it twice. I understand clearly now that pointer variables of increasing indirection must be declared for each reference created.
- Union (only one member is used)
  I didn't know this and I learned it the hard way while working on the project by having a pointer as one member of a union, I had defined, on which delete was called if it was not equal to NULL. If any of the other union members were set the pointer member was not NULL.
- Struct bitfields
  I learned that it is possible to set the maximum number of bits used by members of a struct by placing a colon and integer constant, representing number of bits, after the declaration for the member.

GCC, dlopen API

I became familiar with GCC and some of its options e.g. for setting paths, generating shared or static libraries. I also learned how to use the dlopen API to load shared libraries, to find and call specific functions within them and the significance of functions declared `extern “C”` to avoid C++ name mangling.

Apache Xerces XML library

I became familiar with the Apache Xerces library for reading and writing XML. It is used in the project for reading and writing both the configuration file and test database.

Java, SWT

I gained some experience of using Java and the Standard Widget Tool-kit while writing the plug-in for Eclipse. My attitude towards Java has changed considerably due to this work. In particular, I found it easy to fulfil quite complex requirements in a short space of time with very little debugging required. This was in part due to the language itself and some “spoon-feeding” / hints provided by

the Eclipse IDE which is not available for C++. I intend to study Java further as a result.

Personal

- My professional experience so far has involved working on and extending existing code. Having completed this project from scratch I am much more confident about my abilities. This will assist me in job interviews etc.
- My knowledge of both C and C++ has increased since the start of the year. I'm delighted I have had a further opportunity to study them as I will use both for any personal software projects I undertake.
- In previous years in college I didn't produce adequate documentation for projects etc. largely because I didn't feel it was very important. This year I have gained an understanding of what is required in each document and why they are required. Nearly all of the details of the implementation can be traced back to the Functional Specification and Design Manual. The UML parts of both were especially helpful to me.
- Doing the presentation was an achievement for me as I generally become very nervous with the anticipation of doing anything like it. It provided further confirmation for me that I really enjoy talking about technology.

Review of Project

What went right

Code for walking parse trees

In early March I completely reworked my code for walking the function parse trees produced by Lex & Yacc coupled with my callback functions. Previously this code consisted of a `switch` statement to process the current node, and a loop to iterate through the node's children recursively calling the same code. Walking the trees this way meant setting static `bool` flags to save properties such as `unsigned` which would then be applied to the next type specifier found. I was worried I would reach a point where there would be ambiguity about which item such flags should be applied to. I rewrote the code for walking the parse trees to expect exact sequences of nodes and complain if an unexpected sequence is encountered. To aid the development of this, I created a `printTree()` function which provides a visual text representation of the hierarchy of a parse tree on the console. I have left this function in the library to assist someone updating the grammar at a later stage. The library can be modified to do nothing except provide these visual representations by defining the `VISUALIZE_PARSE_TREE` symbol when compiling the library. As it turned out I was justified in making this change, I would not have been able to support long double, array of pointers, pointer to array or struct/union without it.

Joe Kehoe's multiple tests suggestion

During the project presentation Joe Kehoe suggested that the project should be able to vary the number of sets of test data generated for each function. This feature was included in the Functional Specification and Design Manual, however during implementation I had written a class Test which represented both a test and the function the test was being carried out on – meaning there could only be one test per function without causing problems (e.g. generating CppUnit code, listing tests by

function, grouping tests by function in the test database XML file). Correcting this involved creating a new class to represent each function called Function. The library now returns a list of Function objects to the developer which themselves contain an initially empty list of Test objects. The developer can call `createTest()` on a Function object for each test they want to create. Each of these calls returns a Test object.

Class for XML serialization

I wrote a class called XMLDocument which makes use of Apache Xerces while I was working on the documentation at the start of the year. It is used in the project for both the configuration file and test database.

Constants

In early March I modified the lexer to handle constants. This came about as a result of adding support for arrays. In C, only the first dimension of an array may be specified without a constant (e.g. `char array[][5][5]`) however all dimensions may be specified either (e.g. `char array[5][5][5]`). Having written code for storing a multi-dimensional array as an array with a single dimension I have learned that the size of the first dimension is not required for converting to and from an absolute index. I suspect this has something to do with C/C++ feature just mentioned but don't have time to investigate.

Parameter and TestDatum

There were initially separate classes in the project for representing function parameters (Parameter) and individual test variables (TestDatum). There was a lot of duplication of the code for both of these classes due to the fact that each test variable (TestDatum) is a parameter but with the extra property of containing a value. In mid-March, I removed the duplication by making TestDatum a subclass of the Parameter class. This proved to be even more beneficial when re-factoring the code for serializing to and from XML to allow recursion; both classes can use the same code.

Array of Pointers & Pointer To Array

I discovered that there is a difference between an array of pointers and a pointer to an array. Array of pointers is specified as `char* array[5]`, whereas a pointer to an array is specified as `char (*array)[5]`. I updated the project towards the end of March to make this distinction.

What went wrong

Problems uncovered by the Java VM

The following issues were uncovered by the Java VM and only occurred when the library was being used by the Eclipse plug-in:

- Implicit declarations of callbacks for the Lex & Yacc parser for example an implicit definition of `void* createParseNode()`, but the actual definition is `ParseNode* createParseNode()`. Solution: Rewrite the callbacks to use `void*`, explicitly declare the callbacks at the top of the Yacc grammar and cast to and from `ParseNode*` inside the functions.
- The temporary library generated when getting the initial return value of a function could not

be loaded when the Java VM was running. Solution: Link libstdc++ to the temporary generated library.

- Using the data member of `std::wstring` directly in my `WideToMulti()` function for converting Unicode strings to multi-byte. Solution: Use the `c_str()` method of `std::wstring` to retrieve its contents. This is the correct way to access `std::string` data, my original code was invalid.

Xerces XML parser Malfunction

I had a problem traversing nodes in any reasonably large test database. This issue was present up to and following the Project Presentation. The exact location of the exceptions changed when reading different versions of the test database - but the exceptions always occurred inside the Xerces library. I eventually discovered that the errors were caused by my own code during the loading of an XML document. These are the steps I was performing:

- Instantiate a XercesParser object
- Parse the document
- XercesParser object goes out of scope and is destroyed
- Traverse the document

The issue was that the document object outlived the parser object. It appears that Xerces XML document objects need to continue interacting with their parser object after they have been created. I'm assuming that the reason the errors only occurred on larger documents then is because Xerces does not read XML documents in their entirety but uses buffering.

After correcting the problem above I received no memory errors when testing with Alleyoop/valgrind (a tool for detecting memory errors), where before their were many errors all related to the Xerces XML library.

Integer wrapping on 32-bit Linux

I am developing the software on a 64-bit version of Linux. Towards the end of March I tested the application on a 32-bit version of Linux. The random test values generated on the 32-bit platform were incorrect. The reason for this was that I used `long` for storing the range of `int` values believing that `long` always allows larger values than `int`. This is not the case on 32-bit Linux. `long` and `int` use exactly the same number of bytes and support the same range of values. For this reason my attempt to store the range of `int` values in a `long` resulted in integer wrapping/overflow. I modified the code in question to use `long long` to store the range of `int` values which works on both architectures. Out of interest, here are the sizes of integer and floating point data-types in bytes on 32-bit and 64-bit Linux:

Type	32-bit Linux (bytes)	64-bit Linux (bytes)
short	2	2
int	4	4
long	4	8
long long	8	8
float	4	4
double	8	8
long double	12	16

Table 1: Data-type size differences on 32-bit and 64-bit Linux

Some issues with Eclipse worth mentioning

- After several hours of debugging and writing code with Eclipse CDT it can start to slow up (e.g. windows do not come immediately into focus, delays when typing, indexer and syntax highlighting incorrect). Sometimes it crashes altogether (e.g. Java errors, "unhandled loop exception" was one I received) and requires a restart.
- Debugging with Eclipse is not as easy as with Visual Studio (e.g. the equivalent of Visual Studio's Object Browser in Eclipse is not as sophisticated). Sometimes the debugger cannot retrieve information for certain variables, sometimes it shows the contents of some other variable that you have no interest in. An example of a variable type which cannot be examined while debugging is the std::vector template.
  The upside of this was that I was forced to desk check problematic code.

Export of Eclipse plug-in

I realised on the date of the original deadline that the Eclipse plug-in only worked / was activated when run in the copy of Eclipse launched by Eclipse for debugging Eclipse plug-ins. I didn't think this was due to any errors in my code so I had a look through the plug-in project in Eclipse and discovered the “Export Wizard”. As it turns out, plug-in projects need to be exported using this tool which packages them properly as a JAR file which can be dropped into the “plugin” folder of an Eclipse installation. This is distinct from C / C++ executables which can simply be copied out of the Debug or Release output folders and used.

What I would have liked to include

The following is a list of things I would have liked to implement but could not due to time constraints:

- Support for paths containing spaces (i.e. between quotes)
- CppUnit test file naming specific to each user on the system. At present if two different users on the same system attempt to create tests for the same code stored in the same place on disk the CppUnit files generated for the first user will be overwritten for the second.
- A mechanism which protects the autounit software from serious errors which occur in a user's code (e.g. segmentation fault). A solution might be to run the user's code as a separate process and obtain the data for the result of each function through a pipe. This problem can

present itself when the NULL pointer generation feature is enabled.

- Additional C language support, for example: typedef, global variables, preprocessor statements, external declarations, and type qualifiers: extern and const
- Code parsing, examining the compound statement of each function in order to determine if global variables are used
- Ability, in the Eclipse plug-in GUI, to toggle between NULL pointer and valid pointer for pointer variables in the expected result
- Improved GUI for the Eclipse plug-in

Acknowledgements

Many thanks to Chris Meudec for his help and guidance throughout the year.