User Manual

UCC v.2015.12.2 (Qt 5.7.0 Update)

University of Southern California

Center for Systems and Software Engineering

Version History

Date	Author	Version	Changes
01/23/2007	Marilyn Sperka	0.1	Initial Version
01/30/2007	Vu Nguyen	0.3	Reviewed and updated installation procedures and terminologies to ensure they are consistent with the readme file.
02/03/2009	Vu Nguyen	0.4	Added an instruction to include –DUNIX in the compile line on Unix/Linux.
10/26/2009	Vu Nguyen	1.0	Updated for Release 2009.10
01/12/2010	Marilyn Sperka	1.1	Reformatted and minor edits
06/10/2010	Marilyn Sperka Vu Nguyen	2.0	Updated for UCC v.2010.06
03/10/2011	Marilyn Sperka	2.3	Updated for UCC v 2011.03
05/02/2011	Marilyn Sperka	2.4	Updated for UCC v 2011.05
10/28/2011	Marilyn Sperka	2011.10B	Updated for UCC v2011.10B (B=Beta release)
04/30/2012	Marilyn Sperka Ryan Pfeiffer	2011.10	Updated for UCC v2010.10 release
03/28/2013	Ryan Pfeiffer	2013.04B	Updated for UCC v2013.04B
04/08/2014	Ryan Pfeiffer	2013.04	Updated for UCC v2013.04
08/09/2014	Anandi Hira	2014.08	Updated for UCC v2014.08: Included GUI set up instructions, and GUI specifications
10/16/2014	Tao Li	2014.08	Update for UCC v2014.08: Included handling spaces in path of list file and the path for the file listed in list file.
12/02/2014	Helder Faria e Dias	2014.08	Updated for UCC v2014.08: Included -O3 compilation flag on compilation instructions.
12/05/2014	Helder Faria e Dias	2014.08	Updated for UCC v2014.08: Added –visualdiff switch for visual differencing results.
5/15/2015	Anandi Hira	2015.12	Updated for UCC v2015.12: Made version updates
9/22/2015	Anandi Hira	2015.12	Updated for UCC v2015.12: Added additional features, such as handling prolonged and no extensions in extfile
12/11/2015	Randy Maxwell	2015.12	Updated UCC for v2015.12: Several neew features, improvements, and new reports
12/28/2015	Anandi Hira	2015.12	Updated UCC for v2015.12: Made several formatting changes and added updates.

Table of Contents

1 Introduction

This document provides information for using the UCC tool version 2015.12.

1.1 Product Overview

Most software cost estimation models including the COCOMO® model require some sizing of software code as an input. Ensuring consistency across independent organizations in the rules used to count software cost code is often difficult to achieve. To that end, the USC Center for Systems and Software Engineering (CSSE) has developed and released a code counting toolset called CodeCount to support sizing software code for historical data collection, cost estimation, and reporting purposes. This toolset is a collection of tools designed to automate the collection of source code sizing information. It implements the popular code counting standards published by SEI [1] and adapted by the COCOMO® model [2]. Logical and physical source lines of code (SLOC) are among the metrics generated by the toolset.

Unified Code Count (UCC) is a unified and enhanced version of the CodeCount toolset. It is a code counting and differencing tool that unifies the source counting capabilities of the previous CodeCount tools and source differencing capabilities of the Difftool. It allows the user to count, compare, and collect logical differentials between two versions of the source code of a software product. The differencing capabilities allow users to count the number of added/new, deleted, modified, and unmodified logical SLOC of the current version in comparison with the previous version. With the counting capabilities, users can generate the physical and logical SLOC counts, and other sizing information such as complexity, cyclomatic complexity, comment and keyword counts of the target program.

The UCC tool is provided in C++ source code, and may be used as is, or modified and further distributed subject to certain limitations. The user is responsible for compiling and using the executable version.

2 System Requirements

Note: with large files or baselines, long run time and hanging may be experienced. Refer to Section 9.3 for performance issues.

2.1 Hardware

- RAM: 512 MB. Recommended: 1024 MB. Larger RAM will be able to process larger sets of files.

- HDD: 100 MB available. Recommended: 200 MB available.
- (Optional) Multiple CPU cores for multithreaded capability.

2.2 Operating Systems

- Linux
- Unix
- Mac OS X
- Windows XP/7
- Solaris

2.3 Compilers Supported

- MS Visual Studio 2008, 2010, 2012, 2015
- MinGW
- g++
- Eclipse C/C++
- Any ANSII-Standard C++ compiler RAM: minimum 512 MB. Recommended: 1024 MB
- Qt 5.7.0 and Qt Creator 4.0.2 (if using GUI front-end)
- Note: 16-bit C++ compilers are no longer tested for compatibility

2.4 Additional Software

- (Optional) Boost C++ Library 1.48.0 (if using multithreaded capability)

3 Building UCC Source Code

There is no setup package provided for installing the tool.

3.1 Software Download

The user can download the UCC source files from http://csse.usc.edu/ucc_wp.

3.2 Compilation

The UCC tool can be compiled using an ANSII-Standard C++ compiler. The compiler must support common C++ libraries including IO and STL. UCC is a command-line application, with a GUI front-end, hence, the package contains the code necessary to run UCC as a command-line application, and the code needed to run the GUI front-end. A summary of the directory structure is as follows:

- **ucc<version>**
  - **Counting Rules**: This directory contains the documents of how the languages are being counted, as well as the meaning behind the Cyclomatic Complexity output
  - **gui**: Code specifically needed for the GUI front-end. This code needs to be built in Qt for the GUI to run (which calls the code in the src directory to run).
  - **src**: The main code containing UCC's functionality. This directory alone needs to be compiled to run UCC as a command-line application.

Therefore, using the GUI front-end is optional. UCC can still be run as a command-line application with the code located in the src directory.

3.2.1 Qt for the GUI front-end

UCC is a command-line application, with a GUI front-end constructed with Qt. Qt is compatible with Linux, Unix, Mac OS X, and Windows. The following procedure allows for a user to run UCC using the GUI front-end:

1) Download and install Qt 5.7.0 and Qt Creator 4.0.2 from the following website: http://qt-project.org/downloads

2) If using Cygwin C++ compiler on Windows, rename the Makefile in the ucc parent directory to Makefile_standard, and Makefile_cygwin to Makefile.

3) Run Qt Creator and on the welcome screen, click on the 'Open Project' button. Browse within the *gui* directory within the UCC directory structure, and select the '*gucc.pro*' file.

4) After successful build, a new window will appear with the UCC GUI front-end. All of the functionalities and run options are made available through the GUI.

3.2.2 Visual Studio

On PC based machines, the user can use Visual Studio 2008, 2010, 2012, or 2015 to compile the source code by following the procedure:

1) Create an empty project of Project Type Visual C++ and Template Win32 Console Application. Type in the Project name and Select OK. In the Win32 Application Wizard, select Applications Settings and then select "Empty Project" check box.

2) Select Project/Add Existing Item. Locate and select all UCC source code files in the *src* directory. Click on "Add" to add the selected files. This would add the existing code to the created project.

3) Open the Properties window and go to the Configuration mode page. The user should select "Release" mode for compiling. To choose the mode, Click on Build/Configuration Manager button. Select "Release" from the list "Active Solution configuration". Go to the C/C++ section and click "Precompiled Headers". Make sure the "Create/Use Precompiled Header" selection is "Not Using Precompiled Headers". (*)

4) For Visual Studio 2015, in the Properties, go to Configuration Properties ➔ C/C++ ➔ Preprocessor ➔ Preprocessor Definitions. In the Preprocessor Definitions, add *CRT_SECURE_NO_WARNINGS*.

5) Select Build Solution or use the shortcut Ctrl+Shift+B to compile.

Upon compilation, an executable file will be created in the *Release* folder.

(*) Please note that Visual Studio by default uses the precompiled header option. The user will get error messages if this option is not turned off.

3.2.3 3.2.2.2 MinGW

The following command will compile source files stored in the folder *src* – the below command demonstrates running the command from *src*'s parent directory.

```
g++ ./src/*.cpp -o UCC -DMINGW
```

Note: the command line option -DMINGW must be provided when using the MinGW compiler.

3.2.4 3.2.2.3 g++

The following command will compile source files stored in the folder *src* – the below command demonstrates running the command from *src*'s parent directory.

```
g++ ./src/*.cpp -o UCC -DUNIX -O3
```

Note: the command line option -DUNIX must be provided on UNIX-based systems, including Cygwin and Mac OSX. Furthermore, the command line option -O3 is needed to avoid compatibility issues between Unix-like operating systems.

3.2.5 Boost Library

Note: Boost is not needed for the Qt GUI interface. Users may use multiple threads from the GUI without installing Boost.

3.2.5.1 Compatibility

The library was successfully built and tested on the following platforms:

- Windows XP, Windows Vista, Windows 7 using MSVC 8.0, 9.0, Visual Studio 2010
- Windows Vista using Intel C++ Compiler 10.1.022, 11.1.048
- Mac OS X using GCC 4.2 and newer
- Ubuntu using clang++ 3.5

The following compilers/platforms are not supported by Boost, and may fail to compile the latest Boost library:

- MSVC 7.0 and older
- Borland C++ 5.5.1 (free version). Newer versions might or might not work
- GCC 4.0 and older
- Windows 9x, ME, NT4 and older

3.2.5.2 Installation

1. Download and install/compile Boost Library from: http://boost.org/
    a. On Mac, can optionally install brew: http://www.brew.sh/ . Then from the command line, can install boost, using the following command:
    `brew install boost`
    By default, Boost will be installed in: `/usr/local/Cellar/boost/<version>`
    b. On Linux (Ubuntu), install Boost using the following command from the command line: `apt-get install libboost-all-dev`
    By default, Boost will be installed in: `/usr/include/boost`
    c. On Windows, user will need to compile the Boost libraries. The Boost website can provide further instructions.
2. In UCCThread.h from UCC's src directory, remove the "//" comment symbol from line 39. Lines 38-40 should look like:

```
#else
      // Available if Boost thread library installed, compiled, .h
included and linked.
      #define           ENABLE_THREADS
#endif
```

3. Compile UCC with appropriate commands to link to the Boost library
    a. On Mac: `g++ -stdlib=libc++ -I /usr/local/Cellar/boost/<version>/include –L /usr/local/Cellar/boost/<version>/lib –lboost_thread-mt –`

```
lboost_regex-mt -lboost_system-mt ./src/*.cpp -o UCC -DUNIX -O3 -
pthread
```

b. On Linux (Ubuntu): clang++ -I /usr/include/boost/ -lboost_thread –lboost_regex –lboost_system ./src/*.cpp –o UCC –DUNIX –O3 –pthread
c. On Windows in Visual Studio:

   i. Open the Properties window and go to the Configuration page.
   ii. The user should select "Release" mode for compiling from "Build/Configuration Manager" button. Select "Release" from the list "Active Solution configuration".
   iii. Go to the C/C++ section and click "Precompiled Headers". Make sure the "Create/Use Precompiled Header" selection is "Not Using Precompiled Headers".
   iv. In C++ ➔ General ➔ "Multi-processor Compilation", make sure the option "Yes (/MP)" is selected.
   v. In "Additional Include Directories", add "C:\boost_<version>"
   vi. In Linker ➔ General ➔ "Additional Library Directories", add "C:\boost_<version>\stage\lib".
   vii. Select Build Solution to compile

4. After UCC is built, use the `-threads 2` (or more) command for multiple threads.

4 Running UCC

This document emphasizes how to run UCC from the command line. All of the options mentioned in Section 4.1 are also available from the GUI.

4.1 Command Line Specification Summary

This section describes the command line format used to run UCC, along with a summary of the various switches and their usage. Section 8 will describe each switch in detail along with information on how to tailor the switch usage for various execution requirements as well as performance implications.

UCC [-v] [-d [-i1 fileListA.txt] [-i2 fileListB.txt] [-t #] [-visualdiff]] [-dir <dirA> [dirB] <filespecs>] [-threads #] [-tdup #] [-trunc #] [-cf] [-extfile extFile] [-cc4enable] [-nowarnings] [-nouncounted] [-ramlimit #] [-outdir outDir] [-unified] [-ascii] [-legacy] [-nodup] [-nocomplex] [-nolinks]

`-v`	Displays the version number of UCC being executed.
`-d`	Runs the differencing function.
`-i1 fileListA.txt`	Allows a custom file containing the list of filenames and/or

directories in *Baseline* A to be counted if `-d` is not present or compared if `-d` is present. The file is in plain text format, one filename or directory name per line. If a directory name is specified, the directory is searched recursively for all countable files. If there is a space in the path, add a '\' before the space.

`-i2 fileListB.txt` Allows a custom file containing the list of filenames and/or directories in *Baseline* B. This option can only be used with the -d switch, and the differencing function will be invoked and fileListA.txt and fileListB.txt will be compared. Normally, *Baseline* B is the newer or the current version of the program, as compared to *Baseline* A. If there is a space in the path, add a '\' before the space.

`-t #` Sets the modification threshold, the percentage of common characters between two lines of code to be compared over the length of the longest line. If two lines have the percentage of common characters equal or higher than the specified threshold, they are matched and counted as modified. Otherwise, they are counted as one SLOC deleted and one SLOC added. The valid values range from 0 to 100. If the switch is not used, UCC uses the default threshold of 60 (same as `-t 60`).

`-visualdiff` Enables visual differencing. This causes differences between baselines to be logged in diff_dump.txt and highlighted_diff.html for future visual inspection.

`-dir` Specifies directories containing files to be counted and/or compared. If this argument is provided, the input list files (see above) are ignored. If the `-d` is not present, UCC looks for dirA and filespecs. If the `-d` is present, UCC looks for dirA, dirB and filespecs. The directories are searched recursively for files that match the filespecs. See Section 4.2.1.3 for examples. If there is a space in the path, please add a '\' before the space.

`dirA` Name of the directory. If the option `-d` is provided, dirA is the directory of *Baseline* A. Otherwise, it specifies the directory to be counted with the counting function. If there is a space in the path, add a '\' before the space.

`dirB` Name of the directory of *Baseline* B, used only if the option -d is given. If there is a space in the path, add a '\' before the space.

`filespecs` Specifications of file extensions to be counted/compared; wildcard chars ? * are allowed. Use a space between `filespecs`; for example: *.cpp *.c

`-threads #` Spawns # worker threads. Valid values range from 2 to 80. Requires Boost C++ Library or Qt GUI interface. Note, the multithread capability is not compatible with the `-visualdiff` function.

`-tdup #` Sets the threshold percentage for duplicated files of the same name. This specifies the maximum percent difference between two files of the same name in a baseline to be considered duplicates. The valid

	values range from 0 to 100. By default, this threshold is zero, so the files must be identical by logical SLOC – spacing and comments are not considered.
`-trunc #`	Sets the truncate threshold, specifying the maximum number of characters allowed in a logical SLOC. Additional characters will be truncated. The default value is 10,000, and zero is for no truncation. Performance can be significantly degraded if truncation is too high.
`-cf`	Supports handling ClearCase filenames. The ClearCase application appends version information to the filename, starting from '@@'. This option requires the UCC tool to handle the original filename instead of the ClearCase-modified filename.
`-extfile <extfile>`	Specifies the name and location of a file that contains a map of languages and extensions. This allows the user to include non-default extensions, or remove default extensions. See section 6.1 for examples. If there is a space in the path, add a '\' before the space.
`-cc4enable`	Returns CC4 results (unique conditional clauses for Cyclomatic Complexity). This switch cannot be used with the `–nocomplex` switch.
`-nowarnings`	Removes warning messages from the user interface. The warning messages will still be logged in the log files.
`-nouncounted`	Removes uncounted file messages from the user interface and log files.
`-ramlimit #`	Requires a number that represents 100 MB, which specifies the amount of available RAM for UCC's processing. UCC will then calculate the amount of RAM required for the processing and may give a suggestion if the input is too big for the available RAM. However, the processing will continue. The user may choose to stop UCC's processing manually.
`-outdir <dirname>`	Allows users to specify the location where output reports are to be generated. This allows the user to set up a batch job with multiple UCC runs and not have the output reports overwritten.
`-unified`	Directs UCC to print counting results to a single unified language file name TOTAL_outfile.csv.
`-ascii`	Generate reports in text format. Default is .csv.
`-legacy`	Generate reports in legacy format. Use this option to retain the output files formats supported by Difftool, CodeCount Tools Release 2007.07 and earlier versions. By default, the new output files formats with new fields are applied.
`-nodup`	Does not search for duplicate files. Any duplicates will be reported as unique files. This decreases processing time.
`-nocomplex`	Does not process language keywords or report complexity metrics. Using this switch will reduce the processing time.
`-nolinks`	Skips Unix symbolic links to prevent multiple counting of same files

as links.

Note: If no argument is given the tool reads and counts all files listed in the text file `filelist.txt`. Details on this file are described in Section 4.2.1.1

4.2 Counting and Differencing Details and Examples

4.2.1 Counting Source Files

The counting function is executed using the command line with no `-d` switch. There are two alternative ways to specify the source files of the target program: using the file list (`fileList.txt`) and using the `-dir` switch.

4.2.1.1 Using `fileList.txt`

This command requires the tool to find the file `fileList.txt` in the working directory (`src` directory) and count all source files listed in it:

```
UCC
```

The file `fileList.txt` contains a list of source files to be counted, one filename per line. You can create the file `fileList.txt` using one of the following commands.

- Unix:

  ```
  ls -1 <filespecs> > fileList.txt
  ```

  Or, use the following to obtain full directory/pathname specification:

  ```
  find [Directory] -name '<filespecs>' > fileList.txt
  ```

  To append this file with additional filenames, use:

  ```
  find [Directory] -name '<filespecs>' >> fileList.txt
  ```

- MS-DOS:

  ```
  dir/B > fileList.txt [Directory]\<filespecs>
  ```

  Or, use the following to obtain full directory/pathname specification along with files in all subdirectories:

  ```
  dir/B/S > fileList.txt [Directory]\<filespecs>
  ```

  To append this file with additional filenames, use:

```
dir/B/S > fileList.txt [Directory]\<filespecs>
```

Where,

- `Directory` is the directory name relative to the current working directory (`src` directory). `Directory` is optional, and it is not given, the working directory (`src` directory) is implied. Note that spaces in the directory path are handled.
- `filespecs` is the file specifications, and wildcard chars ? * are allowed. Use a space between two filespecs, for example: *.cpp *.c

4.2.1.2 Specify a custom file list

This command requires the tool to find the file `<fileListA.txt>` in the working directory (`src` directory) and read it to obtain the input source files listed in it. Since the format of these files is the same as `fileList.txt`, you can use the commands described in Section 4.2.1.1 to create them.

```
UCC -i1 <fileListA>
```

An advantage to this method is that the input file list can be given a descriptive name, such as JustCppFiles.txt, and multiple input file lists can be stored in the same directory.

4.2.1.3 Using the `-dir` switch

This command requires the tool to count all source files contained in the folder `project1`:

```
UCC -dir project1
```

This command requires the tool to count all C/C++ source files with file extensions .cpp, .h, .c, .hpp, and .cc contained in the folder `project1`:

```
UCC -dir project1 *.cpp *.h *.c *.hpp *.cc
```

This command requires the tool to difference all source files contained in the folder `project1` with those contained in `project2`:

```
UCC -d -dir project1 project2
```

This command requires the tool to difference all C/C++ source files with file extensions .cpp, .h, .c, .hpp, and .cc contained in the folder `project1` with those contained in `project2`:

```
UCC -d -dir project1 project2 *.cpp *.h *.c *.hpp *.cc
```

Under Unix/Linux when using the `-dir` option, any wildcards must be enclosed within quotes. Otherwise, the wildcards will be expanded on the command line and erroneous results will be produced. For example: `UCC -d -dir baseA baseB *.cpp` should be written as `UCC -d -dir baseA baseB "*.cpp"`.

4.2.2 Differencing Baselines

In this function, source files in the baselines will be matched and compared to determine the counts for logical SLOC added, deleted, modified, or unmodified.

To run this function, UCC must be called with the `-d` switch.

4.2.2.1 Using File Lists

Compare source files of Baseline A and Baseline B contained in files `fileListA.txt` and `fileListB.txt` in the working directory (`src` directory).

```
UCC -d
```

By default, the files `fileListA.txt` and `fileListB.txt` contains source filenames in *Baseline* A and *Baseline* B, respectively. You can specify different filenames by using the `-i1` and `-i2` command line switches.

```
UCC -d -i1 fileA.txt -i2 fileB.txt
```

(Since the format of these files is the same as `fileList.txt`, you can use the commands described in Section 4.2.1.1 to create them).

4.2.2.2 Using the `-dir` switch

This command requires the tool to compare source files of Baseline A and Baseline B contained in directories `project1` and `project2`:

```
UCC -d -dir project1 project2
```

This command requires the tool to compare all C/C++ source files with file extensions .cpp, .h, .c, .hpp, and .cc contained in the folder `project1` and `project2`:

```
UCC -d -dir project1 project2 *.cpp *.c *.hpp *.h *.cc
```

5 Output Files

A variety of output files are produced in order to meet the needs of different types of users. The reports are by default produced in .csv format, which can be opened using Excel where the user can do further analysis. The reports can be produced in text format by using the `-ascii` switch on the command line, and the files will have the extension .txt.

<LANG> is the name of the language of the source files, e.g., C_CPP for C/C++ files and Java for Java files.

File Name	Function	Description
`error_log_<mmddyyyy>_<hhmmss>.txt`	Both	Log file listing errors that occur at the date specified by month, date, year <mmddyyyy> and time specified by hours, minutes, seconds <hhmmss>
`outfile_uncounted_files.csv`	Both	Lists files that were unable to be counted along with the reason (if known)
`UCC_Performance_<mmddyyyy>_<hhmmss>.txt`	Both	Performance information, such as run time, number of files processed, files/second processed
`<LANG>_outfile.csv`	Counting	Counting results for source files of <LANG>
`outfile_summary.csv`	Counting	Summary counting results for all languages and source files counted
`outfile_cplx.csv`	Counting	Complexity results
`outfile_cyclomatic_cplx.csv`	Counting	Cyclomatic complexity results
`DuplicatePairs.csv`	Counting	Lists original and duplicate file pairs
`Duplicates-<LANG>_outfile.csv`	Counting	Counting results for duplicate files of <LANG>
`Duplicates-outfile_cplx.csv`	Counting	Complexity results for duplicate files when counting
`Duplicates-outfile_cyclomatic_cplx.csv`	Counting	Cyclomatic complexity results for duplicate files when counting
`MatchedPairs.csv`	Differencing	Shows how files in *Baseline* A and *Baseline* B were paired for differencing
`outfile_diff_results.csv`	Differencing	Main differencing results
`Baseline-<A|B>-<LANG>_outfile.csv`	Differencing	Counting results for source files of <LANG> for *Baseline* A and *Baseline* B
`Baseline-<A|B>-outfile_summary.csv`	Differencing	Summary counting results for all languages and source files in *Baseline* A and *Baseline* B
`Baseline-<A|B>-outfile_cplx.csv`	Differencing	Complexity results for *Baseline* A and *Baseline* B
`Baseline-<A|B>-outfile_cyclomatic_cplx.csv`	Differencing	Cyclomatic complexity results for *Baseline* A and *Baseline* B
`Duplicates-<A|B>-DuplicatePairs.csv`	Differencing	Lists original and duplicate file pairs in *Baseline* A and *Baseline* B
`Duplicates-<A|B>-<LANG>_outfile.csv`	Differencing	Counting results for duplicate files of <LANG> in *Baseline* A and *Baseline* B
`Duplicates-<A|B>-outfile_summary.csv`		Summary counting results for duplicate files for all languages in *Baseline* A and *Baseline* B
`Duplicates-<A|B>-outfile_cplx.csv`	Differencing	Complexity results for duplicate files in *Baseline* A and *Baseline* B
`Duplicates-<A|B>-outfile_cyclomatic_cplx.csv`	Differencing	Cyclomatic complexity results for duplicate files in *Baseline* A and *Baseline* B

`Highlighted_diff.html`	Differencing	Visual representation of added, modified, and deleted lines between files in *Baseline A* and *Baseline B*

6 Counting Standards

UCC counts physical and logical SLOC and other metrics according to published counting standards, which are developed at CSSE so that the logic behind the metrics being produced is clear to all participants. The counting standard documents are separate documents and are included in the UCC release. The counting standards are derived from the latest available ANSII standard language specification for each language counted by UCC. Users should note that non-ANSII standard compilers might have commands that are outside of the ANSII standard specification. The results from using UCC on non-ANSII standard code cannot be guaranteed.

The counting standards documents for each language provide detailed information of what is counted, and how items are counted, so that all users can understand the operations and outputs of UCC. Definitions are included of what is considered to be a blank line, comment line, and executable line of code for each language. The document describes in detail the physical and logical SLOC counting rules. Physical SLOC are counted at one per line. Logical SLOC counting rules are grouped by structure and the order of precedence is defined.

The items being measured, in order of precedence (numbered), are:

- 1. Executable lines,
- Non-executable lines
    - 2. Declaration (Data) lines
    - 3. Compiler directives
    - Comments
        - 4. On their own lines
        - 5. Embedded
        - 6. Banners
        - 7. Empty comments
        - 8. Blank lines

A table of logical SLOC counting rules is provided, and further specifies the order of precedence for the various types of executable lines. The rules define precisely when a count occurs, and a comments section gives further explanation.

The counting standards define for each language what keywords are counted and tallied in the output report. The keywords include compiler directives, data keywords, and executable keywords. Compiler directives are statements that tell the compiler how to compile a program but not what

to compile. Data keywords define data declarations, which describe storage elements and the format that will be used to interpret data contained in them. Executable keywords are execution control statements. The specified compiler directives, data keywords, and executable keywords are counted and included in output reports. The data keywords are specific to each language.

7 Terminology Explanation

7.1 File Extensions

The tool determines the language of a source file using its file extension. This version supports the following languages and file extensions:

Languages	File Extensions
Ada	.ada, .a, .adb, .ads
Assembly	.asm, .s, .asm.ppc
ASP, ASP.NET	.asp, .aspx
Bash	.sh, .ksh
C Shell Script	.csh, .tcsh
C#	.cs
C/C++	.cpp, .c, .h, .hpp, .cc, .hh
COBOL	.cbl, .cob, .cpy
ColdFusion	.cfm, .cfml, .cfc
ColdFusion Script	.cfs
CSS	.css
Data	Use file mapping with Datafile=<ext>
DOS Batch	.bat
Fortran	.f, .for, .f77, .f90, .f95, .f03, .hpf
HTML	.htm, .html, .shtml, .stm, .sht, .oth, .xhtml
IDL	.pro, .sav
Java	.java
JavaScript	.js
JSP	.jsp
Makefiles	.make, .makefile, (files named Makefile)
MATLAB	.m
NeXtMidas	.mm
Pascal	.pas, .p, .pp, .pa3, .pa4, .pa5
Perl	.pl, .pm
PhP	.php
Python	.py
Ruby	.rb
Scala	.scala

SQL	.sql
VB	.vb, .frm, .mod, .cls, .bas
VBScript	.vbs
Verilog	.v
VHDL	.vhd, .vhdl
X-Midas	.txt
XML	.xml

It may be desirable to associate an additional extension to a language counter, or to disassociate a particular extension from a language counter. This can be done using the `-extfile <filename>` option on the command line. For more information, see Section 8.12.

7.2 Basic Assumption and Definitions

7.2.1 Data Files

Data files shall contain only blank lines and data lines. Data lines are counted using the physical SLOC definition.

7.2.2 Source Files

Source code files may contain blank lines, comment lines (whole or embedded), compiler directives, data lines, or executable lines. Source code files have to be compiled successfully to ensure the integrity of the inclusive syntax.

7.2.3 SLOC Definitions and Counting Rules

Please refer to the counting standard documents.

7.2.4 TAB

A Tab character is treated as a blank character upon input.

7.2.5 Blank Line

A blank line is defined as any physical line of the source file that contains only blank, Tab, or form feed characters prior to the occurrence of a carriage return (EOLN).

7.2.6 Total Sizing

The total sizing of analyzed source code files in terms of the SLOC count contains the highest degree of confidence. However, the sizing information pertaining to the sub classifications (compiler directives, data lines, executable lines) has a somewhat lower level of confidence associated with them.

Misclassifications of the sub classifications of SLOC may occur due to:

1) user modifications to the UCC tool,
2) syntax and semantic enhancements to the parsed programming language,
3) exotic usage of the parsed programming language, and
4) integrity of the host platform execution environment.

Additionally, in some programming languages a single SLOC may contain attributes of both a data declaration and an executable instruction simultaneously. These occurrences represent events beyond the control of the UCC tool designer and may cause the inclusive parsing capabilities of the tool to misclassify a particular SLOC. For these reasons, the counts of sub-classifications should be regarded as an approximation and not as a precise count. In only the physical SLOC definition does the sum of the sub-classification counts equal the total physical SLOC count.

7.2.7 Keyword Count

The search for any programming language specific keywords over a physical line of code for purposes of incrementing the tally of occurrences shall include the detection of multiple keywords of the same type, e.g., two occurrences of the keyword READ on the same physical line.

The search for any programming language specific keywords over a physical line of code for purposes of incrementing the tally of occurrences shall include the detection on multiple keywords of different types, e.g., occurrences of keywords READ and WRITE on the same physical line.

Keywords found within comments (whole or embedded) or string literals shall not be included in the tally count.

8 Switch Usage Detail

This section will describe each switch in detail along with information on how to tailor the switch usage for various execution requirements. When applicable, performance implications will be described as well.

8.1 –v

Displays the version number of UCC being executed.

Initially, the code counting tools were individual counters, and a separate program did differencing. These programs are still available on the public website at http://csse.usc.edu/ucc_wp and are named the CodeCount Tools – Release 2007.07. These tools were combined into a monolith program titled Unified Code Count (UCC). The public website offers multiple versions of UCC. The UCC Title indicates the year and month of the release. For

example, Release 2009.10 was released in October of 2009. Users may choose to use previous releases for reasons that may include there being a bug in a more current release, or a bug was fixed in the current release and the user wants consistency with the way source was previously counted. For releases dated 2011.03 and later, including -v on the command line will cause UCC to print the version number to the standard output device. The message format is "UCC version 2011.03", meaning UCC release 2011.03, or the release of March, 2011.

8.2 –d

If specified, UCC will run the differencing function. Otherwise, the counting function is executed.

UCC can be used to count SLOC within files, but it can also be used to difference two baselines of files. When just counting and not differencing is desired, use the UCC command without the `-d` switch. Counting is the default. When differencing, counting is performed and reported, and additional reports are produced which compare the files in two baselines and determine, for each file, how many logical lines of code were added, deleted, modified, or not modified. For counting and differencing, use the UCC `-d` command.

Differencing is a powerful capability that allows code changes between two baselines (or versions) of code to be monitored. Only differences with respect to logical SLOC are reported to provide insight into the extent of work completed between baselines.

The differencing process matches files between the two baselines using an algorithm that ensures the best possible match is found for all files. The matched file pairs are compared to each other line by line to determine how many logical SLOC have been added, deleted, modified, or are unmodified. All logical SLOC from any files in Baseline A that are not matched to a file in Baseline B are considered deleted, and all logical SLOC from any files in Baseline B that are not matched to a file in Baseline A are considered added.

The user is given the ability to tailor how UCC determines if a logical SLOC has been modified using the `-t #` switch. For more information, see Section 8.5.

The following relationships hold:

Baseline A SLOC = Deleted SLOC + Modified SLOC + Unmodified SLOC

Baseline B SLOC = New SLOC + Modified SLOC + Unmodified SLOC

Differencing will add a significant amount of time to the processing. It also will require more memory. If large numbers of files are in each baseline, there is a possibility that memory will become completely used and the process will hang. This problem may be addressed in a variety of ways.

The user may be able to run the process on a computer that has more memory than the computer that hung, or they may be able to add memory to the computer that hung.

The user may divide the input files into a number of smaller sets. If the user orders the smaller sets by language types, the duplicate file function will still work appropriately. If the files of a language type must be placed in separate sets, the duplicate file function may not find all duplicate pairs. If multiple sets are used, the user may need to aggregate the output files, as this will not be done automatically. The `-outdir` switch may be used to direct the outputs of sequential UCC executions to different directories, enabling the user to more easily specify multiple UCC runs. See Section 8.15 for more information.

The user may choose to disable the duplicate file process by adding the `-nodup` switch to the UCC command line. The search for duplicate files will not be performed, resulting in a performance speedup. This may also result in a reduction of the amount of memory needed, and larger file sets may be possible.

Algorithms that may contribute to the ability of UCC to avoid memory problems are being explored and may be included in future releases.

8.3 –i1 fileListA.txt

Input list filename containing filenames in *Baseline* A.

The `fileListA.txt` file is required to be a plain text format file containing a list of filenames to be processed, one per line. The files may be specified as full or relative directory/pathname specification. UCC will open the file specified by the fileListA.txt argument, read each line and attempt to open each file specified. If a file cannot be opened, an error message is generated, and a tally is kept and reported in the output report.

When counting, as specified by the lack of `-d` switch on the command line, UCC will expect just the `-i1` switch and not the `-i2` switch. When differencing, as specified by the inclusion of the `-d` switch on the command line, UCC will expect to find the `-i2` switch. If these conditions are not met, an error is generated.

8.4 –i2 fileListB.txt

Input list filename containing filenames in *Baseline* B.

If the `-d` switch is present, the differencing function will be invoked and fileListA.txt and fileListB.txt will be compared. Normally, *Baseline* B is the newer or the current version of the program, as compared to *Baseline* A. The file format is same as *Baseline* A.

8.5 –t

Specifies the modification threshold.

The user is given the ability to tailor how UCC determines if a SLOC has been modified using the `-t #` switch. The # is the modification threshold and can be any integer between 0 and 100, with the default being 60 (same as `-t 60`). The modification threshold specifies a percentage; i.e. `-t 60` indicates a modification threshold of 60%. If two SLOC have the percentage of common characters equal or higher than the specified threshold, as compared over the length of the longest line, they are matched and counted as modified. Otherwise, they are counted as one SLOC deleted and one SLOC added. In this example, using `-t 60`, if 60% of the characters of the longest line match with the compared line, the lines are considered modified in Baseline B. If less than 60% of the characters in the longest line match with the compared line, Baseline A counts one SLOC deleted, and Baseline B counts one SLOC added. If the compared lines are exactly the same, the lines are counted as unmodified.

8.6 –visualdiff

Enables visual differencing.

The user is given the ability to visually inspect the differences between Baseline A and Baseline B by using the `-visualdiff` switch. This switch causes the differences between Baseline A and Baseline B to be logged in two different files: *diff_dump.txt* and *highlighted_diff.html*. These files can be later inspected in order to better understand how files from different baselines differ from each other.

8.7 –dir <dirA> [<dirB>] [filespecs ...]

Specify directories containing files to be counted and/or compared.

The `-dir` switch causes UCC to look for files to be counted in the specified directory and its subdirectories. If extensions are provided, it will select only files with those extensions. Any default or specified input filelists will be ignored.

If the `-d` (for differencing) is not present, UCC will only look for <dirA> and filespecs. If the `-d` is present, UCC will look for dirA, dirB and filespecs. The directories are searched recursively for files that match the filespecs.

`dirA` Name of the top level directory. If the option *-d* is provided, dirA is the directory of *Baseline* A. Otherwise, it specifies the directory to be counted with the counting function. The directory search is recursive so that all subdirectories under the top level directory are searched.

`dirB` Name of the top level directory of *Baseline* B, used only if the option *-d* is given. The directory search is recursive so that all subdirectories under the top level directory are searched.

`filespecs` Specifications of file extensions to be counted/compared; wildcard chars ? * are allowed. Use a space between *filespecs*; for example, *.cpp *.c

Examples:

This command will do counting only, since there is no -d, and will look in the directory dirA recursively to find all files. Any files with extensions recognized by UCC will be counted:

```
UCC –dir dirA *.*
```

This command is equivalent to UCC dirA *.*. It will do counting only, and will look in the directory dirA recursively to find all files. Any files with extensions recognized by UCC will be counted:

```
UCC –dir dirA
```

This command will do counting only, since there is no –d, and will look in the directory dirA recursively, and will process all files found with the extensions specified (.c, .cpp, .f, .for, .f77, .f90, .f03, *.hpf). Notice that the file extensions are separated by a space but have no comma between:

```
UCC –dir dirA *.c *.cpp *.f *.for *.f77 *.f90 *.f03 *.hpf
```

This command is improperly formed, as it specifies -d for differencing, but two directories must be provided:

```
UCC -d –dir dirA *.*
```

This command will difference the files found in dirA with the files found in dirB. The *.* direct UCC to process all files with an extension recognized by UCC:

```
UCC -d –dir dirA dirB *.*
```

This command will difference files found in directories dirA and dirB recursively with any extension that UCC recognizes:

```
UCC -d –dir dirA dirB
```

This command will difference files with the extension .java or .sql found in directories dirA and dirB recursively. No other files will be processed:

```
UCC –d -dir dirA dirB *.java *.sql
```

8.8 –threads

UCC will perform faster provided that the underlying hardware has 2 or more CPU cores and the user utilizes the `-threads #` switch. This option requires either the Qt GUI interface or that UCC is linked with a compiled version of the Boost C++ cross platform thread library. Notice, this function is not currently compatible with the `-visualdiff` function.

It may be that using `-threads #` on single CPU core hardware will also run faster. Faster performance for a single CPU core is possible if the optimized build of UCC does not fully use the available CPU to memory, disk I/O, etc. bandwidth.

To use the threads option through the Qt GUI interface, follow the instructions of compiling the GUI in Section 3.2.1.The following gives a summary of how to build UCC linked with Boost C++ Library. http://www.boost.org will have more details.

1. Download and install Boost C++ Library (minimum required version found in Section 2.4)
2. In UCCThread.h from UCC's src directory, remove the "//" comment symbol from line 39. Lines 38-40 should look like:

```
#else

      // Available if Boost thread library installed, compiled, .h
included and linked.

      #define           ENABLE_THREADS

#endif
```

3. Compile UCC with the appropriate commands to link to the Boost library
4. After UCC is built, use the `-threads 2` (or more) command for multiple threads

8.9 –tdup

Specifies the threshold percentage for identical SLOC when comparing two files within a baseline for one file to be considered a duplicate of the other.

This switch specifies the maximum percent difference allowed between the SLOC in two files of the same name within a baseline to be considered duplicates. Blank lines and comments are not considered. By default, this threshold is zero, so the files must be identical by logical SLOC in order to be considered duplicates. Valid values are integers between 1 and 100. The integer corresponds to the percentage of SLOC which may be different when two files are compared in order to be considered duplicates. For example, `-tdup 20` would mean that a file is a duplicate of another file if 20% or less of the SLOC are different.

Duplicate file processing is computationally expensive. A switch -nodup is available to inhibit the duplicate processing in order to reduce execution time.

Files must be in the same baseline, but not necessarily in the same directory, in order to be duplicates. The files do not need to have the same name; however, if the file names are different, they have to be exactly the same, including comments and blank lines, to be identified as duplicates. Files with the same name, but not in the same subdirectory, are considered duplicates if the code is identical, even if there are differences in the comments and/or blank lines.

Duplicate files are counted and reported separately. One purpose for this command is to isolate SLOC counts for files which did not require development, but were duplicated for a variety of reasons which could include for configuration management purposes, or were computer generated.

8.10 –trunc

Truncation threshold.

The truncation threshold specifies the maximum number of characters allowed in a logical SLOC. Additional characters will be truncated. The default value is 10,000. If `-trunc 0` is specified, no truncation is done. Performance can be significantly degraded if truncation is too high.

8.11 –cf

Support handling ClearCase filenames.

The ClearCase application appends version information to the filename, starting from '@@'. This option requires the UCC tool to handle the original filename instead of the ClearCase-modified filename by stripping off the '@@' and any characters after that and before the extension.

8.12 –extfile <extfilename>

UCC uses a file's extension to associates files to be counted with language counting modules. One or more extensions may be associated with a given language counter. The table in Section 7.1 lists the current languages and their associated file extensions.

Using just the `-extfile` switch without specifying the `<extfile>` returns a list of the default languages with related file extensions, which can be used to modify the default file extension settings.

The `-extfile <extfile>` switch allows users to modify the file extension mapping by specifying which file extensions are to be associated with the language counters, including prolonged extensions or no extension. This switch allows the user to include non-default extensions, or

remove default extensions. This gives the user more flexibility in determining which files will be counted. It also will be useful as more languages are added to UCC, as some extensions may be used by more than one language.

Data files will be only be counted for physical SLOC, and only if the user uses an `extfile` to specify the data file extensions. Refer to Section 8.12.2 for more information.

The file named by <extfile> is a list that associates user-specified extensions with UCC language counters. There are no spaces between the language, the equal sign, or the extensions. A comma separates the extensions. Two commas with nothing in between them specifies no extension. Comments may be placed within square brackets, for example `[comment]`, and may placed anywhere in the extfile.

Each line in the file should have the form:

<language>=.<ext1>,.<ext2>,...<extn>

The `-extfile <extfile>` option gives the user great flexibility in tailoring specific runs. For instance, suppose a user wishes to count only the files written in Fortran 77 as indicated by having the extension .f77. Placing the Fortran=.f77 command into the `extfile` would accomplish that. If a user wishes to count Fortran files generated with a non-standard extension, such as .foo. The user can place the command Fortran=.foo in the extfile.

Examples:

This line would cause any file with the extensions .java or .javax, to be counted with the Java language counter. No other extensions will be counted with the java language counter. All other languages will use the default extensions.

```
Java=.java,,.javax
```

These lines would cause the Ada counter to count only files with extensions of .ada or .a. All other Ada files with other extensions will be ignored. Also, the Fortran counter will count only files with extensions of .f or .for. All Fortran files with other extensions will be ignored.

```
Ada=.ada,.a

Fortran=.f,.for
```

8.12.1 File Extension Mapping Names

For reference, the current file extension mappings are shown below. Be sure to use the Internal UCC Language Name in the `-extfile` mapping. For example, don't use C/C++, use C_CPP.

Language	Internal UCC Language Name (for use in the –extfile option)	File Extensions
Ada	Ada	`.ada, .a, .adb, .ads`
Assembly	Assembly	`.asm, .s, .asm.ppc`
ASP, ASP.NET	ASP	`.asp, .aspx`
Bash	Bash	`.sh, .ksh`
C Shell Script	C-Shell	`.csh, .tcsh`
C#	C#	`.cs`
C/C++	C_CPP	`.cpp, .c, .h, .hpp, .cc, .hh`
COBOL	COBOL	`.cbl, .cob, .cpy`
ColdFusion	ColdFusion	`.cfm, .cfml, .cfc`
ColdFusion Script	CFSCRIPT	`.cfs`
CSS	CSS	`.css`
Data	Datafile	Use file mapping with `Datafile=<ext>` (see below)
DOS Batch	DOS_Batch	`.bat`
Fortran	Fortran	`.f, .for, .f77, .f90, .f95, .f03, .hpf`
HTML	HTML	`.htm, .html, .shtml, .stm, .sht, .oth, .xhtml`
IDL	IDL	`.pro, .sav`
Java	Java	`.java`
JavaScript	JavaScript	`.js`
JSP	JSP	`.jsp`
Makefiles	Makefile	`.make, .makefile,` (files named Makefile)
MATLAB	MATLAB	`.m`
NeXtMidas	NeXtMidas	`.mm`
Pascal	Pascal	`.pas, .p, .pp, .pa3, .pa4, .pa5`
Perl	Perl	`.pl, .pm`
PhP	PHP	`.php`
Python	Python	`.py`
Ruby	Ruby	`.rb`
Scala	Scala	`.scala`
SQL	SQL	`.sql`
VB	Visual_Basic	`.vb, .frm, .mod, .cls, .bas`
VBScript	VBScript	`.vbs`
Verilog	Verilog	`.v`
VHDL	VHDL	`.vhd, .vhdl`

X-Midas	X-Midas	`.txt`
XML	XML	`.xml`

8.12.2 Data file counting

The user must specify `Datafile=<ext>` to invoke the data counting function. The tool will output only physical counts and only for the extensions specified.

Examples:

```
Datafile=.dat
```

The reason the `-extfile <extfile>` command must be used for counting data files is that the extension .txt, which is common for data files, was already assigned to the X-Midas counter.

```
Datafile=.dat,.txt
```

8.13 –cc4enable

The 4th and final ring of Cyclomatic Complexity, referred to as CC4 in UCC, identifies identical decision branches with each function. Since the algorithm that identifies identical decision branches requires heavy processing, you may notice that a significant amount of time is required to get the results, and UCC may consume significantly more RAM and CPU resources.

With `-cc4enable`, UCC not only identifies completely identical decision branches, but also clauses that are syntactically identical. For example,

```
if (flag)
```

and

```
if (flag == true)
```

would be identified as identical, and only counted as 1 decision branch. UCC will also identify nested statements that are repeated.

8.14 –nowarnings

The `-nowarnings` switch removes warning messages from the command line and GUI. The warning messages will still be logged in the error log file.

8.15 –nouncounted

The `-nouncounted` switch removes uncounted file messages from the command line and GUI and error log file.

8.16 –ramlimit

The `-ramlimit #` switch requires a number that represents 100MB. Using this representation of how much RAM is available, UCC calculates the amount of RAM is needed for the processing based on the input. UCC will then show a message such as "Information: unlikely to succeed…" and the user may choose to stop UCC before waiting for UCC to run out of RAM and stop with a low on RAM memory message. The valid range of values is from 1 to 5120, which represents 100 MB to 500 GB. Note: Even if UCC calculates that it will not be able to complete the processing with the amount of available RAM given by the `-ramlimit #` command, it will continue to run. The user must stop UCC manually.

8.17 –outdir <dirname>

This command allows the user to specify the location and directory of where the output reports should be generated. If the directory does not exist, UCC will create it. This allows the user to set up a batch job with multiple UCC runs and not have the output reports overwritten.

Examples:

This command will look for the default input list `fileListA.txt`, count the SLOC within the files, and write the output reports into the directory `test1`:

```
UCC -outdir test1
```

This command will open the input list of files in `test2files`, read them in, count the SLOC, and write the output reports into directory `test2`.

```
UCC -outdir test2 -i1 test2files
```

8.18 –unified

Directs UCC to print counting results to a single unified language file name `TOTAL_outfile.csv`.

If the `-unified` command is not specified, a report is produced for each language and is named `<Lang>_outfile.csv`, which is an Excel format.

8.19 –ascii

Generate reports in text format. The default report format is .csv, which opens directly into Excel.

8.20 –legacy

Generate reports in legacy format. Use this option to retain the output files formats supported by Difftool, CodeCount Tools Release 2007.07 and earlier versions. The default format has several additional fields.

8.21 –nodup

Do not search for duplicate files. The duplicate file processing is both memory and processor intensive, as the process analyzes each file and compares to potential duplicates character by character. If the user does not need to separate out duplicate files within a baseline, the `-nodup` switch will disable the duplicate search decreasing processing time significantly. Any duplicates will be reported as unique files.

8.22 –nocomplex

Do not process or report keywords or complexity metrics. Using this switch will reduce the processing time.

8.23 –nolinks

Do not follow symbolic links on Unix based systems. This disables following symbolic links to directories and counting of links to files. This can prevent duplicate file counts on Unix/Linux systems.

9 Performance Issues

Performance is an issue with many facets, among them processor speed, memory size and utilization, bus speed, and program logic. An attempt is made here to point out many of the factors that can affect the performance of running UCC, along with suggestions for improving the performance.

Certain factors, such as the speed of the CPU, or the amount of memory in the computer, are beyond the scope of this document except to point out to your IT manager that it is time for an upgrade. However, we will attempt to point out cases where performance improvements can be made.

9.1 Compiler Optimization

UCC is distributed as open source code. Users download the code and compile the executable using any ANSI-standard C++ compiler. Many compilers offer a variety of optimization levels that

include speeding up the execution of the program, reducing the size of the program in memory, streamlining low level code that is used multiple times, etc. Typically, you gain speed at the expense of space needed for the resulting executable, or you can get a smaller executable that uses less memory, but doesn't run as fast. The user should consult the user's manual for the compiler being used for the appropriate compiler optimization techniques.

9.1.1 Microsoft Visual Studio

In Microsoft Visual Studio, there are options for General Whole Program Optimization, including Use Link Time Code Generation, Profile Guided Optimization – Instrument, Profile Guided Optimization-Optimize, Profile Guided Optimization-Update, and No Whole Program Optimization. Under the C/C++ Optimization men there are options for Minimize Size (/O1), Maximize Speed (/O2), Full Optimization (/Ox), Custom, Inherit from parent or project defaults, or Disabled (/Od). It is left to the user to experiment with the optimization available with the user's compiler in order to meet their particular situation.

9.1.2 GNU g++

The GNU g++ provides a range of general optimization levels, numbered from 0-3, as well as individual options for tailored optimization. The optimization level is specified on the compilation command line with a *-OLEVEL* switch, where *LEVEL* is a number from 0 to 3. The levels are described below.

-O0 or no -O option (default) – No optimization is performed, and the source code is compiled in the most straightforward way. This is the best option to use when debugging.

-O1 or -O – Common forms of optimization are applied that do not require any speed-space tradeoffs. Executables should be smaller and faster than with -O0. Compilation time can actually be less than when compiling with -O0.

-O2 – Includes the optimizations in -O1, plus some further optimizations. No speed-space tradeoffs are used, so the executable should not increase in size. The compiler will require more memory and time for the compilation, but should not increase the executable size. This is the best option to use when producing a deployable program.

-O3 – Includes more expensive optimizations, such as function inlining, as well as the optimizations of levels -O1 and -O2. This option may increase the speed of the executable, but can also increase the size. Under certain circumstances, it might make the program run slower.

-funroll-loops – This option is independent of the other optimization options. It turns on loop-unrolling, and will increase the size of the executable. It may or may not improve speed.

-Os – This option will produce the smallest possible executable, valuable for systems constrained by memory or disk space. In some cases, a smaller executable will run faster, due to better cache usage.

9.2 UCC Switches Affecting Performance

9.2.1 –threads

Internal improvements to UCC for better performance added to this version (about 1.5 times faster than prior 2015 code base). Threads offer even faster performance. Differencing and file reading / analyze / counting will run faster using threads. Tests using 2 extra worker threads on 2 CPU cores hardware gives an overall speed about 2 times faster (rather than 1.5 times). There is a modest RAM overhead for each thread as it uses a thread local private stack instead of the UCC process stack. Users should enjoy faster results using extra worker threads.

9.2.2 –ramlimit

The `-ramlimit` switch gives UCC a memory amount in 100 MB of how much RAM is available for estimation of overall minimum RAM use. The default value is **5** (or 500 MB). Valid ranges are between 100 MB and 500 GB. RAM memory use in this version of UCC has been very much reduced for large sets of files except when wanting to do duplicate checking quickly (see `-nodup` switch description below). UCC internally estimates the amount of RAM needed to do the features like differencing and duplicate checking. UCC uses `-ramlimit` to calculate if the minimum RAM estimate is over the limit then an Information message shows. UCC will still continue with detailed processing.

9.2.3 -trunc

Meaning: Truncation threshold

The truncation threshold specifies the maximum number of characters to be processed in a logical SLOC. Additional characters will be truncated. The default value is 10,000. If -trunc 0 is specified, no truncation is done. Performance can be significantly degraded if truncation is too high. The tradeoff is that when two SLOC are being compared between two programs, the comparison is done character by character, but is stopped when the truncation threshold is met. If there is no difference before the threshold is met, the SLOC are considered unmodified, but if there is a difference after the threshold, the SLOC should have been identified as modified, rather than unmodified. The more long statements (> threshold) the more likely the SLOC identification is to be incorrect. The user must make the tradeoff determination of whether to expend more processing time to process longer statements for greater accuracy, or risk having incorrect SLOC modified/unmodified counts but with a quicker execution speed.

9.2.4 -nodup

Meaning: Do not search for duplicate files.

By default, UCC looks within each baseline for files whose SLOC are identical to the SLOC in other files. The first file is considered unique, but all the other identical files are considered duplicates. Comments and blank lines are not considered in the duplicate processing; i.e. if the comments and/or blank lines change but the SLOC are still duplicates, the file is still considered a duplicate. If the SLOC have been rearranged but no characters have been modified, they are still considered duplicate.

The duplicate files are counted and reported separately from the unique files for the purpose of measuring work done. The Duplicates-<LANG>_outfile.csv, where <LANG> is the language, will have the counts, and the DuplicatePairs.csv file will identify the original and duplicate file pairs.

The duplicate processing is compute intensive, and increases the execution speed. The user can choose to suppress the duplicate processing by using the -nodup switch on the command line. Then all files will be considered unique and will be included in the standard reports.

9.2.5 -nocomplex

Meaning: Do not produce keyword counts or complexity metrics

By default, UCC counts keywords and directive, nested loops, and other metrics useful in evaluating complexity. While this is not a major computational task, some processing time can be eliminated by using the -nocomplex switch to suppress the complexity metrics.

9.3 Large Jobs

Large jobs can be defined many ways. A large job may have large amounts of average size files, or a smaller amount of very large files, or files with very long lines of code. Computers with more memory, more disk space, and/or more processor speed will be able to process larger jobs using less time. This section will give strategies on how to do the best you can with what you have.

9.3.1 Memory Limitations

Symptom: UCC process starts out fine, reporting progress as it goes, and then it hangs. Most likely this is a problem where the process has run out of RAM memory. Since v2015.12, a low RAM memory message will be shown before existing if a memory allocation has failed. The message will also include suggested alternate approaches that are more likely to be successful. Refer to Section 9.2.2 regarding the `-ramlimit` switch, which will allow UCC to warn the user if the estimated minimum needed RAM is more than the given available RAM.

9.3.2 Process One or a Few Languages at a Time

If the input files are written in multiple languages, UCC can be directed to process only certain languages, or even certain extensions within a given language. Refer to Section 8.7 for details on the `-dir` switch and Section 8.12 for details on the `-extfile` switch.

9.3.3 Divide and Conquer

The input files can be regrouped into lesser amounts of files. This can be done by creating extra input file lists, giving each a different name, and dividing up the input files into the various lists. See Section 4.2.1.1 for tips on creating file lists. For example, the user creates 3 file lists called fileList1.txt, fileList2.txt, and fileList3.txt. Each file list can be run separately using the commands:

```
UCC -i1 fileList1.txt

UCC -i1 fileList2.txt

UCC -i1 fileList3.txt
```

Note that the output reports will all be written into the same working directory (`src` directory), and any reports that have the same name will overwrite the previous reports.

9.3.4 Preserving Output Files from Multiple Runs

If multiple UCC runs are to be made consecutively, the user can separate the output report files by using the `-outdir` command.

For example, the following three runs will output reports into different directories.

```
UCC -i1 fileList1.txt -outdir fileList1

UCC -i1 fileList2.txt -outdir fileList2

UCC -i1 fileList3.txt -outdir fileList3
```

UCC will write the output reports into the directory specified by the `-outdir` file. If the output directory does not exist, UCC will create it. Consequently the output reports are segregated by run, and are not overwritten. This allows the user to create a batch job or command file to execute multiple UCC runs sequentially where each consecutive run writes the output reports into a different directory.

9.3.5 Difference in Two Steps

When differencing two large baselines, the user can count each baseline separately to get the duplicate file information. Then difference the two baselines with duplicate processing turned off. See Section 8.21 for more information.

9.3.6 Long Line Truncation

If there are several long lines (greater than 10,000 characters), as indicated by entries in the error log, the user can set the truncation to a smaller number using the switch. Any differences that occur after the truncation will not be detected.

10 Language Specific Information

10.1 Ruby

The Ruby programming language allows continuation characters of "," and ".", but also allows a default continuation if the syntax of a line is not complete. The compiler assumes that the next line is a continuation line. UCC does not examine the syntax. Consequently, if a line is a continuation of the previous line due to context specific information, UCC will not understand that and will count it as a separate line. This could impact the LSLOC count, making it higher than it should be. The PSLOC count will not be affected.

10.2 Fortran

Currently Fortran version 77 and lower uses a fixed format where any character (except 0) in column 6 is a continuation character indicating that this line is a continuation of the previous line. Later versions (F90 and above) do not have this restriction; the continuation character is an & at the end of the line. Currently UCC is using the format for F90 for all Fortran code. There are plans to develop a separate counter for Fortran 77 and lower.

10.3 JavaScript

The JavaScript counter does not count the statement that is not terminated by a semicolon.

11 References

[1] R.E. Park, "Software Size Measurement: A Framework for Counting Source Statements", Technical Report CMU/SEI-92-TR-20 ESC-TR-92-020, 1992.

[2] B. Boehm, C. Abts, S. Chulani, "Software development cost estimation approaches: A survey", *Annals of Software Engineering, 2000.*