Developing and Building CASOS' Construct Simulation Development Environments

Michael J. Lanham, Kenneth Joseph,
Geoffrey P. Morgan, Kathleen M. Carley

December 2014
CMU-ISR-14-115

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Center for the Computational Analysis of Social and Organizational Systems
CASOS technical report.

This work was supported in part by the IRS project in Computational Modeling, the Air Force Office of Sponsored Research (MURI FA9550-09-1-001 mathematical methods for assisting agent-based computation and FA9550-11-1-0179 multi-level cultural models), and the NSF IGERT in CASOS (DGE 997276). In addition support for Construct was provided in part by Office of Naval Research (N00014-06-1-0104 and MURI N000140-81-1-186 a structural approach to the incorporation of cultural knowledge in adaptive adversary models), and the National Security Agency and Army Research Office (W911NF1310154) . Additional support was provided by the Air Force Office of Sponsored Research (MURI N00014-08-1-1186 cultural modeling of the adversary). Further support was provided by CASOS - the Center for Computational Analysis of Social and Organizational Systems at Carnegie Mellon University. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Internal Revenue Service, the National Science Foundation, the Office of Naval Research, the Air Force Office of Sponsored Research, or the U.S. Government.

Keywords: Development Environment, Construct, Integrated Development Environment, Git, Distributed Version Control

Abstract

This technical report provides instructions and guidance to a developer and researcher on how to the setup a development environment to compile, extend, and use Construct. The report provides a complete listing of the tool chain needed in the Windows environment to include Git™, CMake™, Boost, Microsft Visual Studio™. It's primary audience is the CASOS student population, the CASOS staff programmers, and the Carley Tech programming staff. This documentation will reduce the on-ramp time for new developers from several days of discovery learning to two (2) to (4) hours of structured learning.

Table of Contents

1 System Environments

CASOS students have validated this technical report's contents for the following environments:

- Quad-core Intel Xeon with Win 7 Enterprise 64-bit (using CMU's SCS standard image as far as I know), 12GB RAM
- MacBook Pro Core 2 Duo with OS X 10.6.8, Parallels 6 + Windows 7 Professional 32 bit, 4GB RAM
- Most frequently used development environment:
    - Visual Studio 2010 Premium + CMake + Boost Library
    - Visual Studio 2010 Professional + CMake + Boost Library
    - In brief: CMake is a wrapper for make that allows a developer to more rapidly create make files for various environments (e.g., Visual Studio, Cygwin, Linux/Unix).

2 Preparatory Work

2.1 Access to CASOS Git Servers' Projects

Secure access to CASOS Git Servers for the following Projects with a userName and password.

```
userName@hal1.casos.cs.cmu.edu:/usr/git/netstatplus.git
userName@hal1.casos.cs.cmu.edu:/usr/git/eigen.git
userName@hal1.casos.cs.cmu.edu:/usr/git/casos_utils.git
userName@hal1.casos.cs.cmu.edu:/usr/git/tinyxml.git
userName@hal1.casos.cs.cmu.edu:/usr/git/construct.git
```

2.2 3rd Party Libraries and Tools

2.2.1 32-bit Boost

Get the 32-bit c++ boost from http://www.boost.org/. As of August 2014, the up-to-date 32-bit version of the windows installer is 1.56.0 at http://www.boost.org/users/history/version_1_56_0.html. Install Boost following instructions in its Getting Started Guide.

Mike Lanham chose the MS Visual Studio 2010 version, all the libraries (as of 21 Mar 12) and accepted the defaults of the installer from this point forward. The default installation directory was `C:\Program Files (x86)\boost\boost_1_56`

The use of CMake and CMakeFiles configuration files includes the boost libraries (with headers of the format <boost/foo.h>) into the source so the steps below to add its bin directory to your PATH should not be necessary. The steps are in the technical report to facilitate a user doing so should they need to.

1. Control Panel-->System
2. Select `Advanced System Settings` (on the left side of the screen)

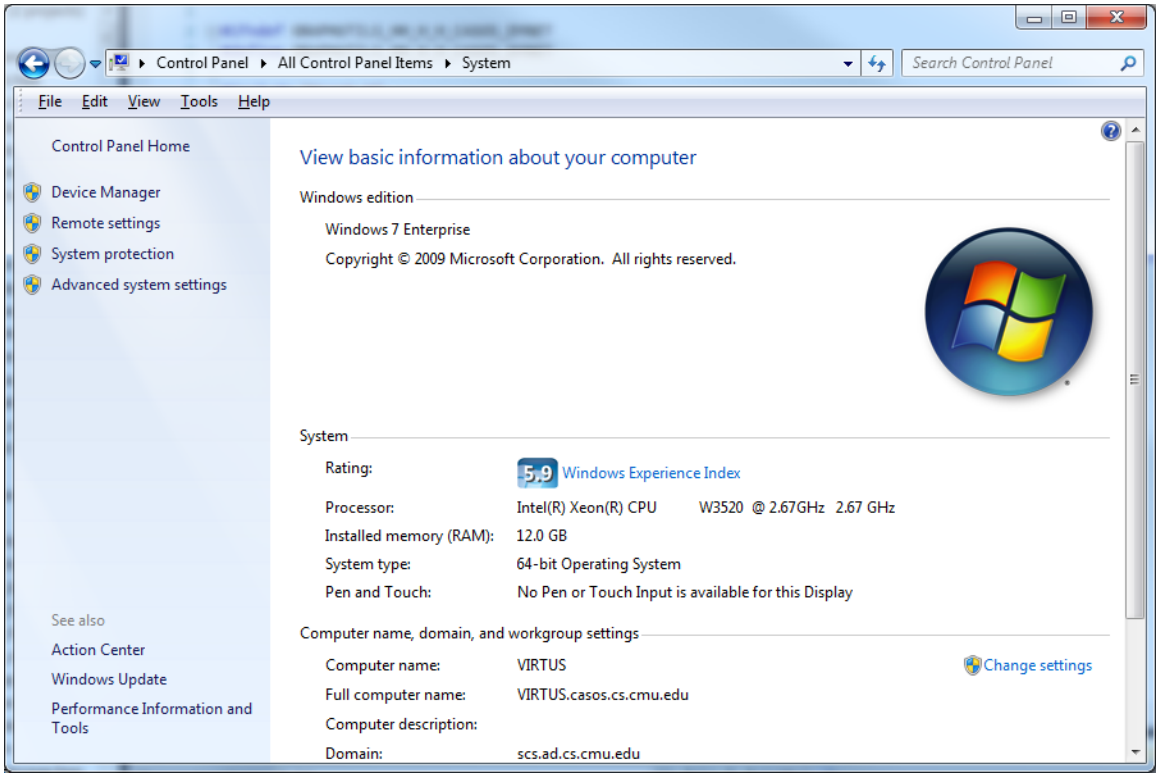

Figure 1 Screen Shot of Windows 7 Control Panel with Advanced system settings displayed

3. Select `Environment Variables` as shown in Figure 2.
4. In the `System variables` portion of the window, select `Path` and `Edit` to add the directory to the path as shown in Figure 3.

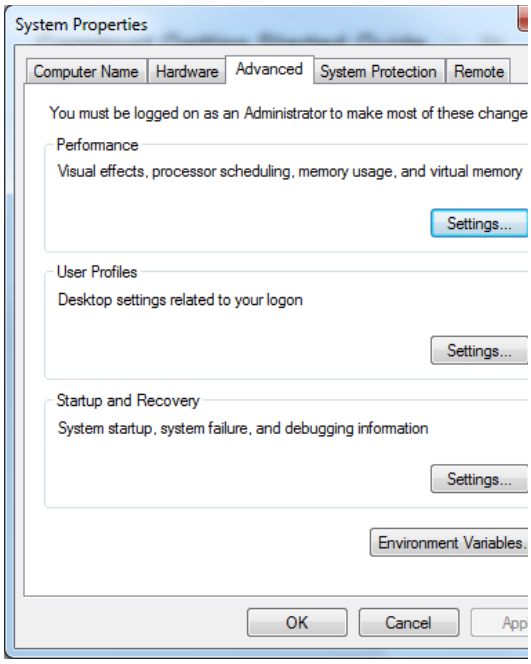

Figure 2 Screen Shot of System Properties Window

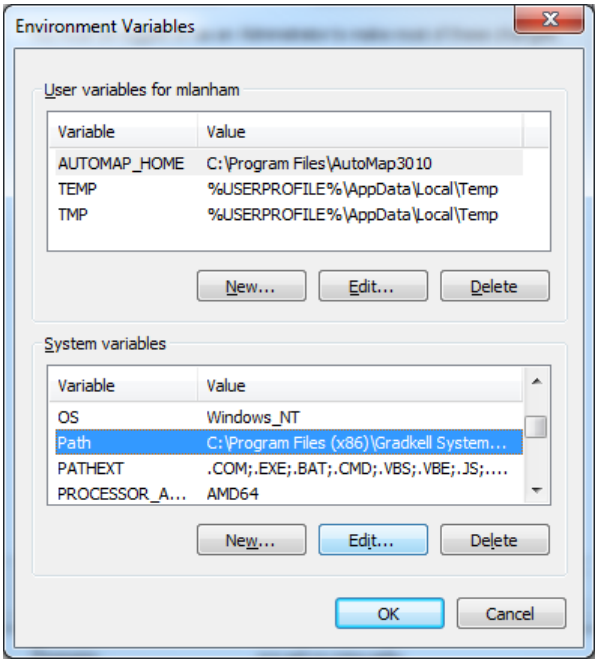

Figure 3 Screen Shot of Environment Variables Window

For other projects, you will likely need to add boost to your MS Visual Studio Projects' settings. The namespace for boost functions/capabilities is boost:: The steps to add boost to your VS project are shown below:

1. Right-click your project in the Solution Explorer pane and select Properties from the resulting pop-up menu
2. In Properties > C/C++ > General > Additional Include Directories, enter the path to the Boost root directory, for example: C:\Program Files\boost\boost_1_56\boost
3. In Configuration Properties > C/C++ > Precompiled Headers, change Use Precompiled Header (/Yu) to Not Using Precompiled Headers.
4. Be sure source code has #include <boost/regex.hpp> (or whatever other header you want to use)

2.2.2 64-bit Boost and Boost Unit_Test

The download above of the 32-bit version includes the source code needed to build 64bit libs.

Open a command prompt *as administrator* and cd into the directory you just unzipped (C:\Program Files (x86)\boost\boost_1_56_0\). At the command prompt type the following commands:

```
.\bootstrap
.\b2 --build-type=complete --without-python --without-mpi
address-model=64 -j5  install
```

This should build the complete boost library, `-j5` will use 4 CPUs (5 threads) to compile, the `address-model` parameter ensures you build a 64-bit version *(the lack of th '--' in front of that parameter is intentional and critical)*. This will also install by default into `c:\boost`. It also installs `c:\boost\include\boost-1_56\boost` and `c:\boost\lib`. The build-type parameter specifies to build both static and dynamic libraries.

If you install elsewhere, be sure to update the CMakeLists.txt file in the experimental_branch to reflect the alternate location. Be Warned: Mike Lanham had zero (0) success getting VS to find the 64-bit boost libraries when I tried to install into `c:\program files (x86)\boost\boost-1_56\lib64` (aka `c:\progra~2\boost\boost-1_56\lib64`).

2.2.3 Version Control and Source Code Management

CASOS has moved to Git from SVN for managing its code. Git is a distributed versioning system, unlike SVN which is a hub-and-spoke versioning system. CASOS had previously been using SVN and encouraged users to install and use TortoiseSVN as their primary client.

If the user/developer has Cygwin installed, it is likely that they also have git installed. If git is not within the Cygwin environment, the developer can re-run the Cygwin setup.exe application to add-to or update the Cygwin environment.

A Windows client for Git is available from http://msysgit.github.io/ (the download page usually lists a number of versions and is located at https://code.google.com/p/msysgit/downloads/list?q=full+installer+official+git). Install this software accepting the provided defaults. This installer will install two (2) flavors of clients, one for command line and one GUI.

Alternatively, or in addition to Git for Windows, you can install TortoiseGit from https://code.google.com/p/tortoisegit/ (more accurately from https://code.google.com/p/tortoisegit/wiki/Download). TortoiseGit will provide windows explorer based capabilities to interact with Git.

A Macintosh OS X Git client available in the Apple ™ App store is called SourceTree and is free to download and install.

2.2.4 Microsoft Visual Studio 10

Download and install Microsoft Visual Studio 10 or newer from whatever source is legitimately available to the reader.

2.2.5 CMake

Download and install CMake from http://cmake.org/cmake/resources/software.html. Mike Lanham used the following directory for installation of `C:\Program Files (x86)\CMake_3.01`.

2.2.6 Doxygen

Get and download Doxygen from http://www.stack.nl/~dimitri/doxygen/download.html. Mike Lanham accepted the default directory, and by default, that dir is added to the `PATH` environment variable.

As of August 2014, the current version is 1.8.7.

2.3 Build Development Directory Structure

1. Within your `Visual Studio 2010\Projects` directory, create a directory called `CASOS_code`. Within this directory you will create four (4) sub-directories.
2. Create a sub-directory named `casos_utils`.
3. Create a sub-directory named `construct`.
4. Create a sub-directory named `eigen`.
5. Create a sub-directory named `netstatplus`.
6. Create a sub-directory named `tinyxml`.

Figure 4 below is a screen shot of the directory structure you should have built.

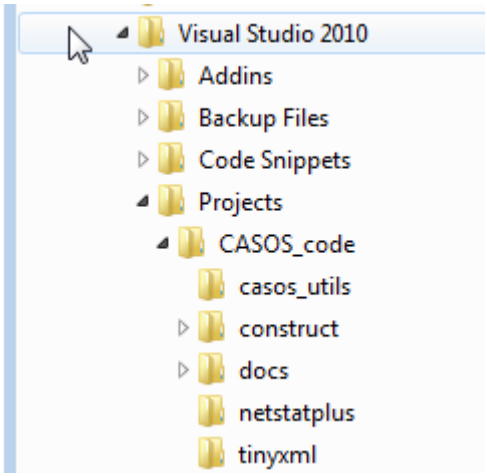

Figure 4 Screen Shot of Development Directory Structure

2.4 Checkout Development Source Code

For each directory in `CASOS_Code`, clone the four (4) repositories listed in Section 2.1. The easiest way to do this is to use the Windows command line. As of August 2014, Git GUI for Windows ver. 0.19.GITGUI does not appear to support specifying which branch to check out.

As a duplicable example, to check out the experimental branch of the tinyxml source code, open a Windows Command prompt from the Windows Start menu (for Win 7 and below), open the Git Bash command prompt. See also Figure 5.

Change directories using the cd command to the directory structure you created in section 2.3.

```
cd “documents\Visual Studio 2010\Projects\CASOS_code”
```

Still using the command line version of git, clone from the CASOS master repository to a local repository and specify the `CMU` branch for each cloned repository as shown in the example below. After prompting for a password, and assuming entry of a valid password for the valid user identity, git will download the specified branch to the specified directory.

```
git clone --progress -v
mlanham@hal1.casos.cs.cmu.edu:/usr/git/casos_utils.git -b
CMU .\casos_utils
```

The `-b` switch specifies the branch to clone. See also Figure 6. As of August 2014, the password the git server expects is the your School of Computer Science (SCS) Kerberos password. Do *not* use your Windows™ Active Directory™ (AD) SCS password.

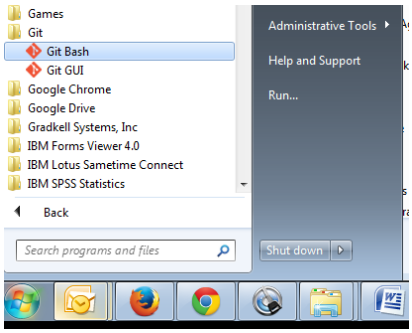

Figure 5 Start Git Bash command prompt

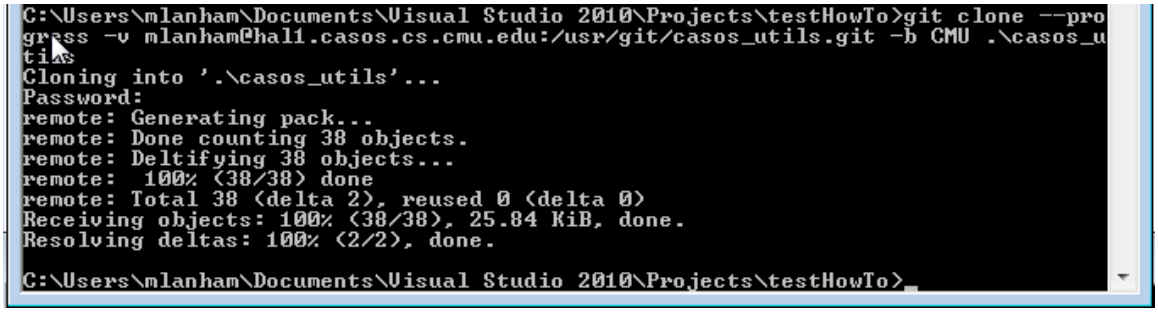

Figure 6 Git clone command line of specified branch to specified directory

It is now feasible for the user to shift to the Git GUI for Windows if the user desires a GUI instead of a CLI. To do so, select the “Git GUI” icon from the windows start menu (Win 7 and earlier).

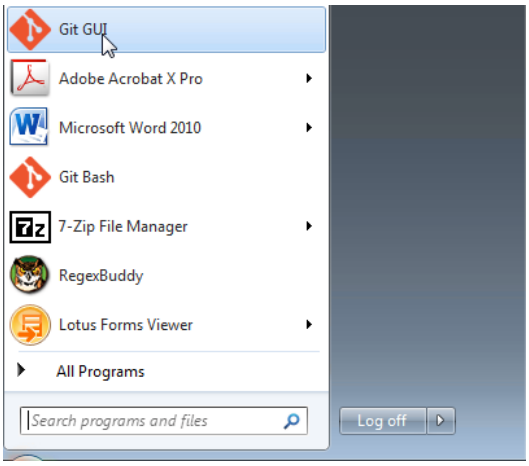

Figure 7 Select Git GUI from Windows Start Menu

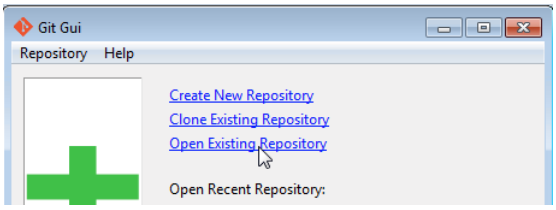

Figure 8 Select Open Existing Repository from Git GUI

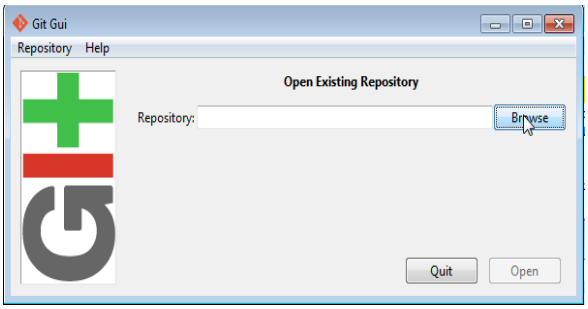

Figure 9 Browse to the cloned repository directory

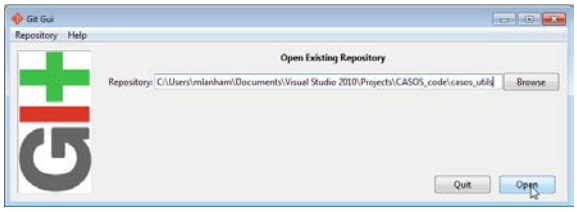

Figure 10 Open Selected Repository

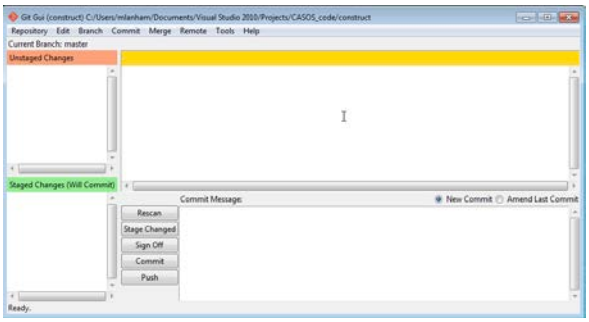

Figure 11 Git for Windows GUI view of Construct repository

2.4.1 CASOS' Basic Development and Source Code Revision Workflow

CASOS is using the Git Feature Branch Workflow for Construct described at the following URL: https://www.atlassian.com/git/workflows. For users familiar with the centralized version control methods of SVN, this is very similar process.

The most basic code authoring process follows the Centralized Workflow process using the list of steps below. When developing new features for the Construct code base, researchers should use the Feature Branch discussed in Section 2.4.2.

1. Clone repository (akin to SVN checkout)
2. Commit local changes to local repository as development proceeds
3. Decide development changes need to return to the central server and the `master` development line (in this exemplar, the central server is `hal1.casos.cs.cmu.edu`). Now follow the sub-process below:

1. Rebase the local copy using the following command to pull changes from the master server to the local repository. This command and process is akin to a SVN Update where a developer pulls changes from the repository before committing their own changes. This increases the likelihood that the developer will notice and resolve conflicts before attempting to commit their own changes.

```
git pull --rebase origin master
```

2. Resolve any existing conflicts between the `master` line on the central server and the local `master` development line.
3. Send the local changes to the central repository using the following command.

```
git push origin master
```

> *A set of diagrams that depicts this centralized workflow is at the following URL: https://www.atlassian.com/git/workflows#!workflow-centralized.*

2.4.2 CASOS' Feature Branch Workflow with Git

When developing new features for the Construct code base, researchers and developers should use the Feature Branch discussed at the following URL: https://www.atlassian.com/git/workflows#!workflow-feature-branch. This workflow takes advantages of a few of Git's strengths. The intent is to have researchers and developers author new features of construct within their new-feature branches on personal systems. This isolation of developmental code from the main development line reduces the probability of an author pushing their incomplete code into the master development line on the primary server.

The steps below presume that a developer has already cloned the master development line from the central repository.

1. Create an isolated development branch using the command below

```
git checkout –b my-feature-branch master
```

2. Commit local changes to local repository as development proceeds
3. Decide development changes need to return to the central server (in this exemplar, it remains `hal1.casos.cs.cmu.edu`), though not yet to the `master` development line. This step facilitates the sharing of the specialty code with other developers that are collaborating on the feature development. Use the command below the first time a developmental branch returns to the central server.

```
git push –u origin my-feature-branch
```

4. Subsequent pushes of my-feature-branch use a shorter version of the command as shown below.

```
git push
```

5. CASOS does not operate an automated mechanism to have one developer send a `pull-request` to other developers. However, when a contributing developer is satisfied that the functionality within the my-feature-branch is ready for incorporating back into the 'master' development line, they can send an email to the person(s) CASOS designates as the quality assurance/quality control (QA/QC) point of contact (POC). The email can answer the 5 W's (i.e., who, what, why, when, where) and can form the basis of the commit log in the `master` development line.
6. The QA/QC POC will follow the four (4) step sequence of steps below to integrate the `master` development line and the `my-feature-branch`.

```
git checkout master
git pull
git pull origin my-feature-branch
git push
```

This process will cause the QA/QC POC's git client to shift to the `master` development line. Then retrieve any updates to that line from the central repository. Then pull the changes in `my-feature-branch` into the `origin` development line. And finally to push those changes from the local environment back to the central repository server.

> *A set of diagrams that depicts this centralized workflow is at the following URL: https://www.atlassian.com/git/workflows#!workflow-feature-branch.*

3 Build Construct Libraries and Executable

3.1 CMake and build directories

Within each `src` directory, there is a `CMakeList.txt` configuration file. The developer can run cmake from the command line, or allow Visual Studio to automatically recognize the file has changed and re-run cmake from within Visual Studio. As configured, CMake is intended to create compiled make files for libraries and executables out-of-source (in directories segregated from source files, and in Construct and its supporting libraries, at the top of the directory tree for each project/library).

3.1.1 CMake configuration files

As stated above, within each `src` directory, there is a `CMakeList.txt.` That configuration file may also include or references other configuration files. This is especially true for the construct repository.

Within the construct/src directory there are, as of August 2014, five (5) CMake files. Each file name and description is in Table 1. It is very important that the CMakeLists.file_list.txt file be a complete and up-to-date list of header (.h), source (.cpp) and other files necessary for the project. CMake has no other way of being aware of which files are part of the project!

Table 1: CMake files for construct repository

File Name	Purpose
CMakeLists.txt	Primary configuration file that CMake.exe expects to find. Invoking this file, triggers the entire tool chain to start, process file lists for changes and possible recompilation, generate compiler options etc.
CMakeLists.file_list.txt	The file that contains the lists of header and cpp files to include into the Visual Studio or other development environment's list of applicable project files. **Critical**: CMake has no way of knowing about any other files in a project except through this file. Users must add file names to this file as hey author them!
CMakeLists.bsd.txt	Used by cmake when the researcher is using a BSD development environment or compiling for the BSD runtime environment. **Important:** any BSD-specific headers(.h) or source (.cpp) should be included in this file by appending them to the lines that concatenate to the variable created in the file_lists.txt file.
CMakeLists.win.txt	Used by cmake when the researcher is using a Windows development environment (in this TR, the environment is Microsoft's Visual Studio 2010).
CmakeLists.SharedLibs.txt	Facilitates the compilation into .dll libraries instead of compiling static libraries.

3.1.2 CMake's build directories

There should be four (4) build directories for each project that are permanently part of the repository you cloned. A list of the four build directories is below:

1. `buildWin32`

2. buildWin64
3. buildBSD32
4. buildBSD64

A full path for the tinyxml compiled libraries for Windows 64-bit libraries for user mlanham is shown below.

```
C:\Users\mlanham\Documents\Visual Studio
2010\Projects\CASOS_code\tinyxml\buildWin64
```

Each of these build directories should have an explanatory `README.TXT`, to remind the user how to invoke cmake within that build directory.

> *CMake does not support (as of August 2014) the ability to have a single build directory and invocation to support both 32-bit and 64-bit target platforms in the same `.sln` file.*

3.2 Use of CMake and 64-bit Windows compilation preparation

1. Open a command prompt and navigate to the `CASOS_code` directory.
2. Now navigate further to the `casos_utils\buildWin64` directory. This is CMake's sandbox directory...the `CMakeLists.txt` file, when executed as specified below, creates Visual Studio project and solution files here, an intermediate directory for object files. The directory also becomes the home for 64-bit compilations (e.g., debug, release) you select from within Visual Studio's IDE.

> *Do not execute the steps in a cygwin window the first time CMake is invoked during any particular boot-cycle Doing so can lead to a set of errors from Visual Studio's MSBuild that will end with something similar to the one below.*
>
> *1>C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\Platforms\Win32\Microsoft.Cpp.Win32.Targets(147,5): error MSB6001: Invalid command line switch for "CL.exe". Item has already been added. Key in dictionary: 'TMP' Key being added: 'tmp'*
>
> *This is due to Cygwin's environment variables being case sensitive (e.g. `tmp != TMP`) whereas Windows environment variables are not case sensitive (e.g. `tmp==TMP`). If you get this*

error, delete the x64 directory, reboot, make a new x64 directory, and run CMake from a CMD prompt instead. This error is persistent until the computer is rebooted!

After reboot, a user could also `unset TMP` in a Cygwin environment before running cmake within Cygwin. This will prevent MSBuild from seeing two versions of the `tmp` environment variable.

3. Type the CMD shell commands below. This style of invoking cmake without prepending its path information is why we added CMake's bin directory to the system's PATH environment variable. If you choose to not modify the PATH environment variable, prepend the necessary path information in the invocation to the example below.

```
cmake -D USE_X64=true -G "Visual Studio 10 Win64" ..\src
```

7. After cmake completes, without errors, double click the `.sln` file to open it with Visual Studio.
8. Using the menu bar, select Build-->Build Solution
9. Repeat this process for the listed repositories that are part of the Construct application (i.e. `casos_utils, netstatplus, tinyxml`). There is nothing to 'make' with Eigen so there is no need to run cmake on it, despite netstatplus' extensive use of Eigen.

If you get errors about not finding boost when compiling construct....open and edit the `construct\src\CMakeList.txt` to ensure the path you installed boost in is the path reflected in the `CMakeList.txt`.

The steps above should have compiled each of three compilable libraries. If there was a failure or error, building one library at a time will reduce the space within which troubleshooting must occur

10. It is now time to compile the construct library and link it with the libraries we verified compiled in the steps above. Now navigate further to the `casos\buildWin64 directory.`
11. Type the CMD shell commands below.

```
cmake -D USE_X64=true -G "Visual Studio 10 Win64" ..\src
```

These last two steps should, in sequence, re-create a Visual Studio project for each of the libraries construct depends on. It should also create a Visual Studio project for the Construct project.

From this point, skip to Section 3.4 for instructions on building and linking each supporting library and linking those libraries to form an executable.

3.3 Use of CMake and 32-bit Windows compilation preparation

1. Open a command prompt and navigate to each library's `buildWin32` directory.
2. Type the following

```
cmake -G "Visual Studio 10" ..\src
```

> *Note the lack of the 'Win64' inside the quotes for the command above, telling cmake to configure the Visual Studio to use the 32bit compiler instead of the 64 bit compiler. Also, note the lack of the –D USE_X64 option. That command-line variable helped control internal-to-CMakeFile control logic and compiler switches passed to the Visual Studio Compiler.*

12. After cmake completes, without errors, double click the `.sln` file to open it with Visual Studio.
13. Using the menu bar, select Build-->Build Solution
14. Repeat this process for the listed repositories that are part of the Construct application (i.e. `casos_utils, netstatplus, tinyxml`). There is nothing to 'make' with Eigen so there is no need to run cmake on it, despite netstatplus' extensive use of Eigen.

> *If you get errors about not finding boost when compiling construct….open and edit the `construct\src\CMakeList.txt` to ensure the path you installed boost in is the path reflected in the `CMakeList.txt`.*

The steps above should have compiled each of three compilable libraries. If there was a failure or error, building one library at a time will reduce the space within which troubleshooting must occur

15. It is now time to compile the construct library and link it with the libraries we verified compiled in the steps above. Now navigate further to the `casos\buildWin32 directory.`
16. Type the CMD shell commands below.

```
cmake -G “Visual Studio 10” ..\src
```

These last two steps should, in sequence, re-create a Visual Studio project for each of the libraries construct depends on. It should also create a Visual Studio project for the Construct project.

The project will allow the user to use Visual Studio to do the following:

- See each library’s headers and source files
- Compile all the libraries, link the link the libraries into the dynet executable
- Compiled the project’s unit tests and linked those with boost’s unit test framework
- Run the project’s unit tests.

17. Now open the construct Visual Studio project file by double clicking on `construct.sln` within the `buildWin32` directory.

3.4 Use of Visual Studio to build and link Construct

The Visual Studio project file and Integrated Development Environment (IDE)will allow the user to use Visual Studio to do the following:

- See each library’s headers and source files
- Compile all the libraries, link the link the libraries into the dynet executable
- Compiled the project’s unit tests and linked those with boost’s unit test framework
- Run the project’s unit tests.

1. Open the construct Visual Studio project file by double clicking on `construct.sln` within either the `buildWin32` or the `buildWin64` directory.
18. We must now set the startup project, the project that has the main(), for this solution file. Unfortunately, this cannot be automated using CMakeLists.txt as Visual Studio stores the setting in a binary file that CMake cannot access (as of July 2013). Right click on Solution ‘construct’ at the top of the Solution explorer as depicted in Figure 12. Then select the `Properties` menu item to have Visual Studio open the properties window as shown in Figure 12.

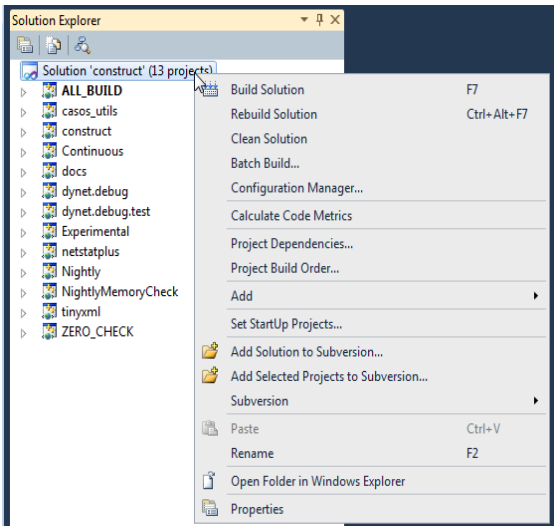

Figure 12 Results of right clicking "Solution 'construct' (13 projects)"

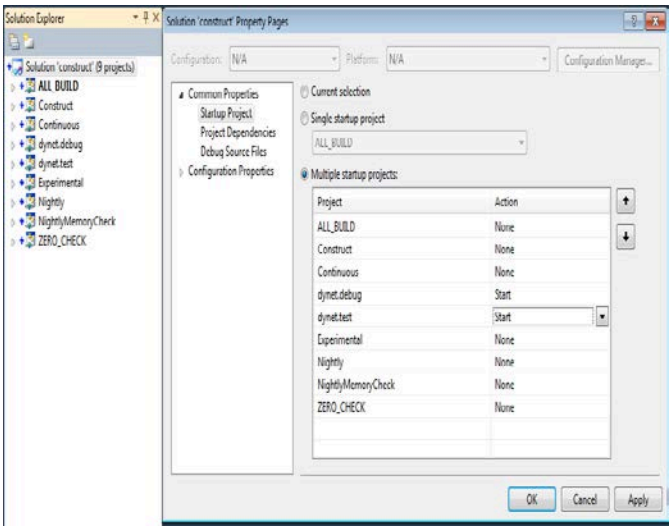

Figure 13 Screen Shot of Construct Project Properties Window

If you want ONLY the dynet.debug (the debuggable Construct) to run, then Select Single Startup project and change the startup project to dynet.debug.

If you want ONLY the dynet.test (Unit test framework executable) to run, then Select Single Startup project and change the startup project to dynet.test.

If you want two or more projects to start when pressing F5 (Debug), then Select "Multiple startup projects:" and change dynet.debug and dynet.test 'Action' from "none" to "start." Figure 13 illustrates this setting.

19. Now compile the entire solution by using the Build→Compile Solution menu item. There should be no fatal errors. There will likely be a number of warnings and other messages in the console output screen during compilation.

After the steps listed in Section 4.2, post compilation should also show in the console output the execution of the unit tests as a post-build event.

3.5 Use of CMake and 64-bit BSD compilation

The scripts supporting the creation of makefiles for BSD are in need of refinement and testing to ensure as-expected performance as of July 2013!

1. Open a command prompt and navigate to each library's `buildBSDxx` directory.
2. Type the following

```
cmake –D USE_X64=true –D BSD=true -G "Unix Makefiles"
..\src
```

3. Type `make all`

3.6 Use of CMake and 32-bit BSD compilation

1. Open a command prompt and navigate to each library's `buildBSDxx` directory.
2. Type the following

```
cmake –D BSD=true -G “Unix Makefiles” ..\src
```

3. Type `make all`

4 Unit Tests for Construct

4.1 CMake configured Unit Tests

Using the `construct/src /CMakeLists.txt` and CMake from the command prompt will pre-configure unit tests and their execution after each build of `dynet.debug.test`.

> *Boost automatically creates a main() method so you will NOT find such a method in the tests directory. Boost does this during pre-compilation resolution of conditional statements and macros. Specifically the BOOST_TEST_MAIN macro creates the main() function.*

> *Unit tests are in the construct/tests directory. As of July 2013, there are very few classes with any tests in place.*

Boost's Unit Test Framework (UTF) is not overly complex, and potentially not compatible other development environments.

4.2 Running Construct Unit Tests

There are two ways of invoking the unit tests within Visual Studio.

1. Select `Dynet.debug.test` and press F5, which should start an instance of `dynet.debug.test`
2. Right click dynet.test and select properties (should already be done by CMakeLists.txt). Then select Linker --> System --> SubSystem choose from the list option Console (/SUBSYSTEM:CONSOLE). Also select Build Events --> Post-Build Event (should already be done by `tests\CMakeLists.txt`). For Command Line enter the value below

```
"$(TargetDir)\$(TargetName).exe" --result_code=no --
report_level=short
```

Build Events > Post-Build Event > Description

```
==== Run unit tests ====
```

To run tests in a Unix environment (according to CTest web pages) type

```
make test
```

4.3 Add new test case(s) to the test harness

Update your SVN repository...you should pull a new sub-directory called tests under the experimental branch we have been working on

The DynetTest.cpp is the main program in this testing harness. For each class (the plan as of 4 April 2012), there will be a `ClassToBeTestedTest.cpp` file and possibly a `ClassToBeTestedTest.h`

Create the empty files in the file system!

Add the file names to CMakeLists.txt!

You now have a choice

Re-run CMake from a cmd prompt (aka DOS Command Window). For 64 bit testing, within the `experimental_branch/x64` directory, type

```
cmake -D USE_X64=true -G “Visual Studio 10 Win64” ..
```

Press Build in Visual Studio to invoke make. You will then get a series of windows and messages that files have changed outside the environment. That is normal, and you will want to re-open the files in Visual Studio as prompted by the window alerts.

Without this re-build, you will not see your new file (and header) in the Visual Studio environment. Once you see the file in the IDE, you can begin adding test cases as your test design specifies.

Each .cpp file will need the setup and teardown ‘fixture.’ You can read more about CMake fixtures in CMake’s documentation.

Each .cpp file will also have the various test suite(s) [a logical container for test cases] for the class under test, as well as the test cases [a logical container for test assertions].

To incorporate the newly written tests, you must rebuild `Dynet.debug.test`. Build the test files in the usual way (Build Menu-->Build Solution or F7)

Run the tests via gui (Debug Menu-->Start without Debugging) or via command line.

4.4 MS Visual Studio Test Professional

Don't bother with Construct. Visual Studio's ability to help developers with unit tests is constrained by Microsoft (and the nature of the C Language) to using "managed code" [Code that uses the common language runtime (CLR) (i.e., .NET) framework. Since CASOS develops and uses Construct in Linux and Windows environments, a "managed code" set of unit tests would not be feasible. The sections above with installing and configuring VS 2010 for use with Boost's unit_test framework should, ultimately, work on both development platforms and in both execution environments.

5 Profiling Construct

5.1 "Very Sleepy" Profile Tool

Download the profiler http://www.codersnotes.com/files/verysleepy_0_82.exe. Though at least one student developer, Mike Lanham, installed and began initial exploration of this product, this portion of the technical report remains a task-to-be-done.

A quick note about the difference between inclusive and exclusive columns in tool: *inclusive* means the total amount of time spent in function while *exclusive* means the amount of time spent in function minus any time spent calling other functions.

More information needed!

5.2 Microsoft's Profile Tool

Since we are not installing MS Visual Studio Team or Professional editions, VS does not come equipped to profile applications. MS has made available a standalone profiler available at the link below to install.
http://www.microsoft.com/download/en/details.aspx?id=23205

Though at least one student developer, Mike Lanham, installed and began initial exploration of this product, this portion of the technical report remains a task-to-be-done.

More information needed!

6 Conditional Compilation & Pre-Processor Statements

There is a `README.h` file in the source file folder that captures, as of July 2013, all the conditional compilation statements in use by Construct. Included below is the content of that file. Where there is a series of three question marks (???) it reflects the authors current ignorance of the exact function of the conditional compilation.

#define	Purpose	Where Used
`__GNUC__`	Supports tinyXML's efforts to be cross-compiler usable	Tinyxml.h (tinyxml)
`_CORES_DEBUG`	???	
`_MSC_VER`	disables certain warnings when developer is using MS	Tinyxml.h (tinyxml)

	Visual Studio IDE when USING_DB is defined links mysql's 64bit ODBC library (in dbClient.h)	
BOOST_THREAD	???	construct.cpp
DEBUG	enables code segments dedicated to debugging	many places
DEBUG_PROBABILITY_OF_INTERACTIONS	prints to stdout additional information on how construct calculates probability of interaction as well as who is interacting with whom on a per interaction basis	InteractionManager.cpp
DEBUG_INTERACTIONS	prints to stdout additional information about who is interacting with whom	InteractionManager.cpp
DEBUG_TM	prints to stdout additional information on how construct is caculating knowledge transactive memory updates to agents as well as ego's core activations scores for alters, alters' groups, and ego's groups	Agent_TM.cpp ???
DEBUG_PARSER	supports debugging of TinyXML's parser	tinyxmlparser.cpp (TinyXML)
DEFAULT_HEADER		
EIGEN_INITIALIZE_MATRICES_BY_ZERO		Matrix_core.h (netstatplus_
GPTL	???	ConstructNetwork.cpp
IRS	???	for use in supporting IRS-specific-project
macintosh	???	
NO_AP_ASSERT	???	netstatplusap.h
NOMINMAX	Suppresses other definitions of MIN and MAX	Cu_mutex.h (casos_utils)
NONONO	allows developer to compile construct in GUI mode (see also main.cpp for linker instructions)	

THR	???	cu_mutex.cpp
TIXML_SAFE	Supports tinyXML's efforts to be cross-compiler usable	Tinyxml.h (tinyxml)
TIXML_SNPRINTF	Supports tinyXML's efforts to be cross-compiler usable	Tinyxml.h (tinyxml)
TIXML_SSCANF	Supports tinyXML's efforts to be cross-compiler usable	Tinyxml.h (tinyxml)
TIXML_USE_STL	support's TinyXML's use of STL	TinyXML.h (tinyxml)
TypeDefException		
UNIX	incorporates unix operating system specific code to take advantage of] windows specific capabilities (or sometimes to prevent breaking the cross-compilation for other operating systems)	Multiple files (casos_utils,
USE_ATLAS	???	netstatplus matrix.h
UseExceptions		
USE_NSP	incorporates netstatplus code which is the core of CASOS' ORA network analytic software. This code is **essential** for the printing of DynetML formatted output	Construct module
USE_THREADS	supports multi-threading of construct initialization of interaction probability matrix	InteractionManager.cpp
USING_DB	incorporates code required for construct to accept inputs and outputs to/from ODBC databases. Code primarily written for MySQL connectivity as of Aug 2012	
WIN32	incorporates windows operating system specific code to take advantage of] windows specific capabilities (or sometimes to prevent breaking the cross-compilation for other	Multiple files (casos_utils,)

	operating systems)	