AutoTester: Introduction and Usage

Your SPA implementation will be tested using an automated tester called **Autotester**. The AutoTester is implemented in C++ and is in the form of a static library that should be linked with your SPA to make a separate application. In order to run the AutoTester, ensure that you have installed Visual Studio 2015. More details and instructions are below.

Preparing for AutoTester

To enable your AutoTester:

1. Download the zip file containing all the necessary Autotester files. It includes these files:
   - AutoTesterInUse.sln: Click on the .sln file to open the solution in your Visual Studio IDE.
   - AutoTesterLib.lib: This is a static library containing the object (.obj) files that will be subsequently linked with the object files of your SPA. Use this when you are compiling your solution in "Release" mode.
   - AutoTesterLib_debug.lib: This is a static library containing the object (.obj) files that will be subsequently linked with the object files of your SPA. Use this when you are compiling your solution in "Debug" mode.
   - AbstractWrapper.h: This is a header file that must be placed in the same directory as TestWrapper.h. **Please do NOT modify this file** (otherwise, it will cause problems for AutoTester later).
   - TestWrapper.h: This is a header file containing the specification of a TestWrapper class that is in charge of the communication between the AutoTester and your SPA.
   - TestWrapper.cpp: This file will contain the implementation of the various methods of the TestWrapper class. The implementation of these methods will be done by you. Comments are included in the file to guide you.
   - Release/Sample-Source.txt: A sample SIMPLE source program for you to test your TestWrapper implementation.
   - Release/Sample-Queries.txt: A set of queries to go with the sample source program. Note that this is just a sample set of queries. The source program and queries used in the final testing will be more complicated than this! Also note that this set of test queries may include queries that you may have not implemented yet. Please remove such queries accordingly. The file specifies test cases - five line for each case: test case ID with comments, declaration, query, expected result, and allowed time in milliseconds.
   - Release/analysis.xsl: After a successful run of Autotester, it will generate a .xml file. Open this XML file in Mozilla Firefox. Mozilla Firefox will automatically convert the xml file to an html file using analysis.xsl.
2. To hook up to the AutoTester, you only need to implement the TestWrapper class. The following is the source of TestWrapper.h:

```
#ifndef TESTWRAPPER_H
#define TESTWRAPPER_H

#include <string>
#include <iostream>
```

```
#include <list>
#include "AbstractWrapper.h"

// include your other headers here

class TestWrapper : public AbstractWrapper {
public:
// a default constructor
TestWrapper();

// method for parsing the SIMPLE source
virtual void parse(std::string filename);

// method for evaluating a query
virtual void evaluate(std::string query, std::list<std::string>& results);
};

#endif
```

First, in the header file (TestWrapper.h), include any other header files necessary for compiling the TestWrapper class. If you need to use instance variables in your implementation, create a private section in the class specification and declare your instance variables there. Next, you need to implement the 3 methods specified in TestWrapper.h by writing code for the methods in TestWrapper.cpp.

The first method is the default constructor. In this method, you can create objects that you will need later on for parsing and query evaluation. You can also do any initialization your need for your SPA here. The second method is called parse. The AutoTester will pass the name of the SIMPLE source file through parameter "filename" of this method. What you need to do here is to write code that will invoke the parser/design extractor of your SPA to create the PKB components from the source file.

The third method is called evaluate. It has 2 parameters: query will contain the query string while results will store the answers to the query. Here, you will write code that will invoke your SPA to evaluate the query. The answers to the query will be subsequently stored in the results list.

The **PQL query comes in the form of a single line** (Even though it is given as two lines in the test case file - those two lines are concatenated by the AutoTester with a space in the middle). For example, "assign a; stmt s; Select <a, s> such that Parent(s, a)". There are only spaces between the declarations and the query (not a newline). On the other hand, answers to the query must be stored as std::strings in the results list. For example, if the answer to the query is 1 2, 1 5 and 1 8, results will contain 3 elements: "1 2", "1 5" and "1 8". We shall now discuss the different formats of the returned results:

- Select BOOLEAN ......
  For boolean results, the results list contains only one element - "true" or "false", with all characters in *lowercase*.
- Select a such that ....

For example, if your answer is 9, 7, 1 and 3, then the results list will contain 4 string elements: "9", "7", "1" and "3". Note that you only need to return the statement number(s) if the synonym in the Select clause is of type statement, assign, while or if. If the synonym is of type variable or procedure, return the variable and procedure names respectively.

- Select <p, s> such that ...
  For tuple results, each result is stored as a string in the results list. However, each component of a result must be separated by **one space**. For example, if the results are <p, 1>, <q, 4>, then the results list will contain 2 elements: "p 1" and "q 4". Notice that there is only one space between p and 1 as well as q and 4 (two or more spaces are not allowed).
- Select s.stmt#... or Select <p.procName, v.varName>...
  The format of the results is similar to the previous two cases. For example, if the results to "Select s.stmt#..." are 1 and 6, the results list will contain 2 elements: "1" and "6".
- If there are *no* results, just leave the results list empty. Don't add anything to the results list, not even an empty string.

Important: The order of the results is not important. The AutoTester will do the sorting of the results for you. The AutoTester will also remove any duplicates it finds in the answers. For example, if you return the answers "1", "3", "3", "2", "3", the AutoTester will consider your answers to be just "1", "2" and "3". Note also that when the result is a procedure name or variable name, make sure you do not change the case. For example, if the result is procedure First, it should be returned as "First", not "first" or "FIRST" or "FiRSt" etc.

3. The AutoTester works in a *cooperative* mode. In other words, instead of forcing your program to stop whenever the time limit is reached during the evaluation of a query, it will first signal to your program to stop. Then, it waits for a couple of seconds for your program to do any clean up e.g. deallocating memory. If your program returns within the extended time limit, the AutoTester will continue with the next query. Otherwise, it will print out a message to inform us that the program has refused to terminate, in which case we will forcefully terminate the whole testing and give a full penalty for that query.

   It is important that you return the goodwill of the AutoTester by responding to the stop signal. We will heavily penalize programs that refuse to cooperate with the AutoTester. To do the checking of the stop signal, you need to add some code into your own SPA program as sketched out in the following pseudocode:

```
#include "AbstractWrapper.h"

.....

for each query_clause in query do
  evaluator.eval(query_clause)
  if (AbstractWrapper::GlobalStop) {
    // do cleanup
    ....
    return
  }

.....
```

First, ensure that you add the AbstractWrapper header file Thread.h using '#include "AbstractWrapper.h"'. To check for the stop signal, you check the static global variable AbstractWrapper::GlobalStop which is a bool value. If the value is true, it means that the time has expired for the query and you should stop execution and do any cleanup and return control back to the AutoTester. Otherwise, you are still within the time limit and you can continue processing the query.

You can do the checking at more than one place in your program. For example, the evaluation of Affects* is often time consuming. So, you might want to do a check for every 50 answers found during the evaluation of an Affects*. Since every team's implementation is different, you have to judge for yourself where are good points in your program to do this checking. Generally, you will want to do checking in *strategic* places e.g. after the evaluation of a query clause. However, note that checking takes time. Therefore, too much checking may slow down your system. There is no hard and fast rule on how much checking is required. You have to find a suitable level that will strike a balance between the efficiency and cooperativeness of your implementation.

Compiling and running the AutoTester

Once you have implemented your TestWrapper class and added the necessary checking code in your SPA program, it is time to compile the AutoTester with your SPA program. Simply use Visual Studio automatic build tool (press F7).

Important: The 'main' method is located in the AutoTester. Therefore, you should not compile a source file that contains the method main. Either remove it or comment it out.

To run the AutoTester, enter the following command from the command prompt at the release directory:

```
AutoTester Sample-Source.txt Sample-Queries.txt out.xml
```

The AutoTester accepts three arguments. The first is the name of the file containing the SIMPLE source. The second is the name of the file containing the queries. The third is the output file to store the results of the testing.

Alternatively, you may run AutoTester with additional arguments:

1. `AutoTester Sample-Source.txt Sample-Queries.txt out.xml -f Query-Id`

   '-f Query-Id' runs AutoTester from a specific Query-Id onwards from the queries file ('Sample-Queries.txt')

2. `AutoTester Sample-Source.txt Sample-Queries.txt out.xml -n Query-Id -t Time-Allowed`

'-n Query-Id -t Time-Allowed' runs AutoTester for a specific Query-Id in the queries file ('Sample-Queries.txt'). The query evaluation is stopped after 'Time-Allowed' miliseconds

Once the testing is completed, you can view the results of the testing by opening out.xml in Internet Explorer (IE).

Important points

For your final submission, make sure your SPA can communicate with the AutoTester as described above. The Auto-testing procedure done by us will be similar except that we will be feeding the AutoTester with our own set of source program and test queries.

There are some important points you need to take note of:

1. The AutoTester program make use of the namespace AutoTester. Therefore, if you are using namespace in your program, you should avoid calling your namespace AutoTester.
2. Please follow the answer format strictly. For example, if you're returning tuple results such as "1 2", "3 4" etc, make sure that you don't have excessive spaces in-between. Answers such as "1   2" will be regarded as wrong by the AutoTester. This will in turn result in unnecessary time waste when we double-check the failed cases.
3. During the implementation of your TestWrapper, you might want to print some debugging information to the standard output. This is allowed but disable those std::cout statements in your final submission. Printing debugging information during the testing will slow down your SPA and result in unnecessary timeouts (each query is only allowed a certain amount of time to complete)
4. Please compile your program in Release configuration. By default, AutoTesterInUse solution uses Release configuration. Mostly due to inappropriate coding practice, one might find problem/bugs when switching between Debug and Release configuration. Release configuration performs optimization, while Debug doesn't.
5. The SIMPLE source file used for the final testing will be about 500 lines. The number of test queries are also around 500. So, please do extensive testing of your SPA before final submission. You should also try to run your own SPA for a large set of consecutive queries say 100-200 queries on a specific SIMPLE source file to see whether your SPA is robust enough to withstand large number of queries without crashing.
6. AutoTester is not 'open-source'. Please do not request for the source code for AutoTester.
7. You are welcome to use AutoTester for your own system testing. But the main purpose of AutoTester is final testing, for which it has been proved to be adequate. Therefore, no requests for enhancements please!