20 October, 2014

National University of Singapore

CS3202 Software Engineering Project

ITERATION 3 REPORT

Team Number: 4 Consultation Day/Hour: Tuesday, 1pm

Team Name: Team 4

Team Members Information:

Group PKB

Saloni Kaur	A0084053L	a0084053@nus.edu.sg
M I Azima	A0085594N	a0085594@nus.edu.sg

Group PQL

Saima Mahmood	A0084176Y	a0084176@nus.edu.sg
Nguyen Trong Son	A0088441	A0088441@nus.edu.sg
Vu Phuc Tho	A0090585X	A0090585@nus.edu.sg

1. Iteration Overview

1.1. Scope of SPA Implemented

In this second iteration we have followed the suggested implementation of the SPA according to the assignment document. We have extended the functionality of the PKB to include the Affects relationship. The QP has been extended to deal with tuple results. The Query Evaluator has been optimized as well.

1.2. Achievements & Problems

The main achievement for this iteration would have to be the fact that we managed to implement the requirements expected from this iteration. We also implemented a design extractor to ease the functionality from the parser. Furthermore we also decided to change the underlying data structure used to construct all of the PKB tables for the various relations. This will be discussed further in depth in section 4.

From iteration two we picked up on some problems of our system, with the building of the next and affects relations. This, will be further elaborated on in section 8.

2. Project Plans

The way that we have decided to break down the project into tasks is shown in tables 1 and 2 below.

2.1. Plan for Whole Project

The breakdown of tasks reflected in table 1 is tentative and will be subjected to change as we move along the various iterations. The subsequent changes, if any, shall be reflected in the reports of the corresponding iteration.

	Iteration 1				**Iteration 2**			**Iteration 3**		
Team Member	PKB	Parser	Query Processer	Report	PKB	Query Processer	Report	Affects Relationship	Tuple Results	Report
Azima	*			*	*		*			*

Saima			*	*		*	*			*
Saloni	*			*	*		*			*
Sean		*		*		*	*			*
Tho			*			*	*			*

Table 1: Whole Project Tasks Breakdown

2.2. Plan for Iteration 3

Team Member	Testing	Writing Test Cases	Revamp of Relationships Data Structure	Affects	Working on QP	Fixing Issues with QE	Report
Azima	*	*					*
Saima	*				*		*
Saloni	*	*					*
Sean	*		*	*			
Tho	*				*	*	

Table 2: Iteration 3 Work Distribution

3. UML Diagrams

SAME AS BEFORE

4. Design Decisions

In this section we will discuss the data structure we have used in representing the relationship tables in the PKB. The implementation of Affects/Affects Star and Next Star will also be discussed. The implementation of the Design Extractor will also be discussed.

4.1. PKB Tables' Data Structure

In this iteration of our SPA development, we have decided to revamp the way in which we represent the PKB tables. It has to be noted that previously in iteration 2, the tables were implemented via 2D vectors storing Boolean values. Fig. 1 shows a representation of the *Follow* table as was previously implemented.

	4	5	6
4	0	1	0
5	0	0	1
6	0	0	0

Figure 1: Previous Representation of Follows Table

However we realized that when the size of the table would have to be extended the time complexity to resize the table would be O(n^2). As such we decided to change this to an implementation where we do not have to resize the tables.

Now we have decided to replace all of the 2D vector representations with a MapTable, for all relationship tables, and a ListTable for the remaining tables such as the Symbol Table. The ListTable has been implemented by a single vector, while the MapTable consists of 4 Map attributes. All of this is displayed in fig 2.

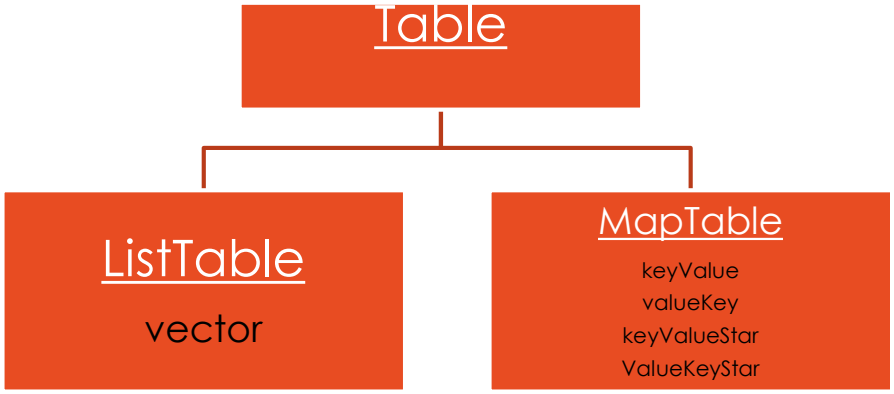


Figure 2: ADT used for PKB

As can be seen from the figure, the 4 attributes for MapTable will store the various relationship information in the form of a key-value of a map. To capture the 2 way relationship of any entity encountered, we have implemented 2 maps, keyValue and valueKey, to keep track of the forward and backward direction relationship. keyValueStar and valueKeyStar similarly keeps track of the corresponding star relationships.

To give an example, the implementation of the Next Table could look like as follows.

MapTable <int> NextTable;

keyValue (Map of type <int, vector<int>>)

Possible value pairs: (1,2), (2,3), (3, 4), (4, 5)

valueKey (Map of type <int, vector<int>>)

Possible value pairs: (2,1), (3,2), (4, 3), (5, 4)

keyValueStar (Map of type <int, vector<int>>)

Possible value pairs: (1, [2, 3, 4, 5])

The reason for keeping track of the backward direction relationship is to ease the computation of the queries when the 1st or 2nd argument is unknown. Based on the unknown value, the PKB will access the relevant maps.

4.2. Affects/ Affects Star

For the implementation the Affects relationship, and the star relationship by extension, we have added methods to determine Affects in the PKB. Since affects is calculated on the fly during the query evaluation, Affects does not have a table per say. Based on a given query, the QE will call a method in the PKB to determine the Affects/Affects star relationship. Then in the PKB, the corresponding method will do the following:

i. First of all check if the two arguments are in the same procedure. This will be done by a breadth first search in the CFG paths of the 2 arguments. The CFG is basically the Next Table.
ii. The next thing to occur is that all possible CFG paths will be discovered.
iii. After that, for every CFG path the method will check if between the start and end program line of this CFG path, a variable is not modified in any assignment statement or procedure call
iv. A list of results will then be returned to the QE to be returned to the main controller.

Affects star also works similarly by calling the relevant affects methods in the PKB in a loop, until all the possible answers that satisfy the query are found.

4.3. Design Extractor

As part of the earlier iterations of the SPA we did not deem it necessary to implement a Design Extractor, DE, since the Next, Affects relationships were not to be implemented. However as part of this iteration, it was necessary to have included a DE to ease the functionality of the Parser. The flow of the SPA, with the inclusion of the DE, now works in the manner as shown in fig 3.

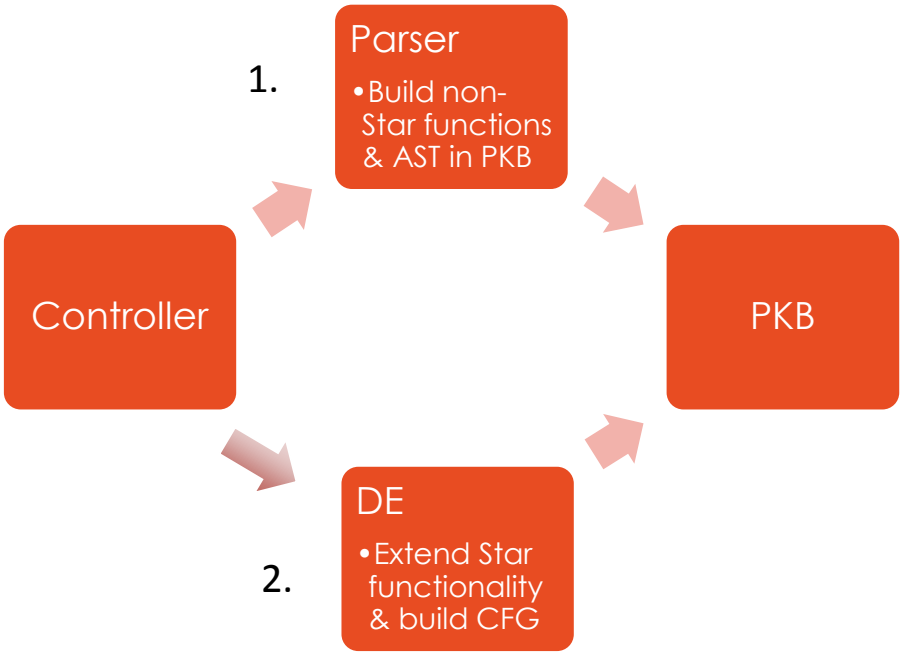


Figure 3: Parser & DE Program Flow

As seen from fig. 3, we can see how the DE takes up more of the complicated functionality of implementing the relationships in the PKB. In step 2, the DE also completes the Modifies and Uses tables for procedure calls and Modifies/ Uses statements.

5. Coding Standards & Experiences

SAME AS BEFORE

6. Query Processing

In this section we will go through how queries are evaluated. For the discussion below we will be referring to the sample query, shown below, to illustrate the process.

stmt s1, s2, s3;

Select s1 such that Parent*(s1, s2) pattern s3(_, _"x+y*2"_)

6.1. Query Validation

The query validating process is handled by the QueryPreprocessor (QPP). The QPP will read and validate data from the query file, and store all necessary information for query evaluation in the Query Representator (QR).

During validation, the QPP checks each query line by line, following these rules:

- On finding a line containing a declaration (e.g. stmt s;), QPP will send this part to the `preprocessDeclaration()` method.
- On finding a line containing a query part (e.g. Select s such that Follows(s, 1)), QPP will send this part to `preprocessQueryPart()` method.
- While the `preprocessQueryPart()` method runs, if the QPP finds a new clause of query, it will call the corresponding method `preprocessClause()` (e.g. `preprocessSuchThatCondtion()`) for this clause. Currently, we have provided the methods for "*such that*" clause and "*pattern*" clause.

After validating the query, if no error is found, the QPP will save validated data in the QR. The QR will contain the query tree. An example of what the query tree, based on the sample query, will look like is shown below.

6.2. Query Evaluation

The query evaluation process is carried out by the QE. Using the stored data from the QR and PKB, the QE will try to find values that satisfy the query's conditions, and send the list of result values to the AutoTester.

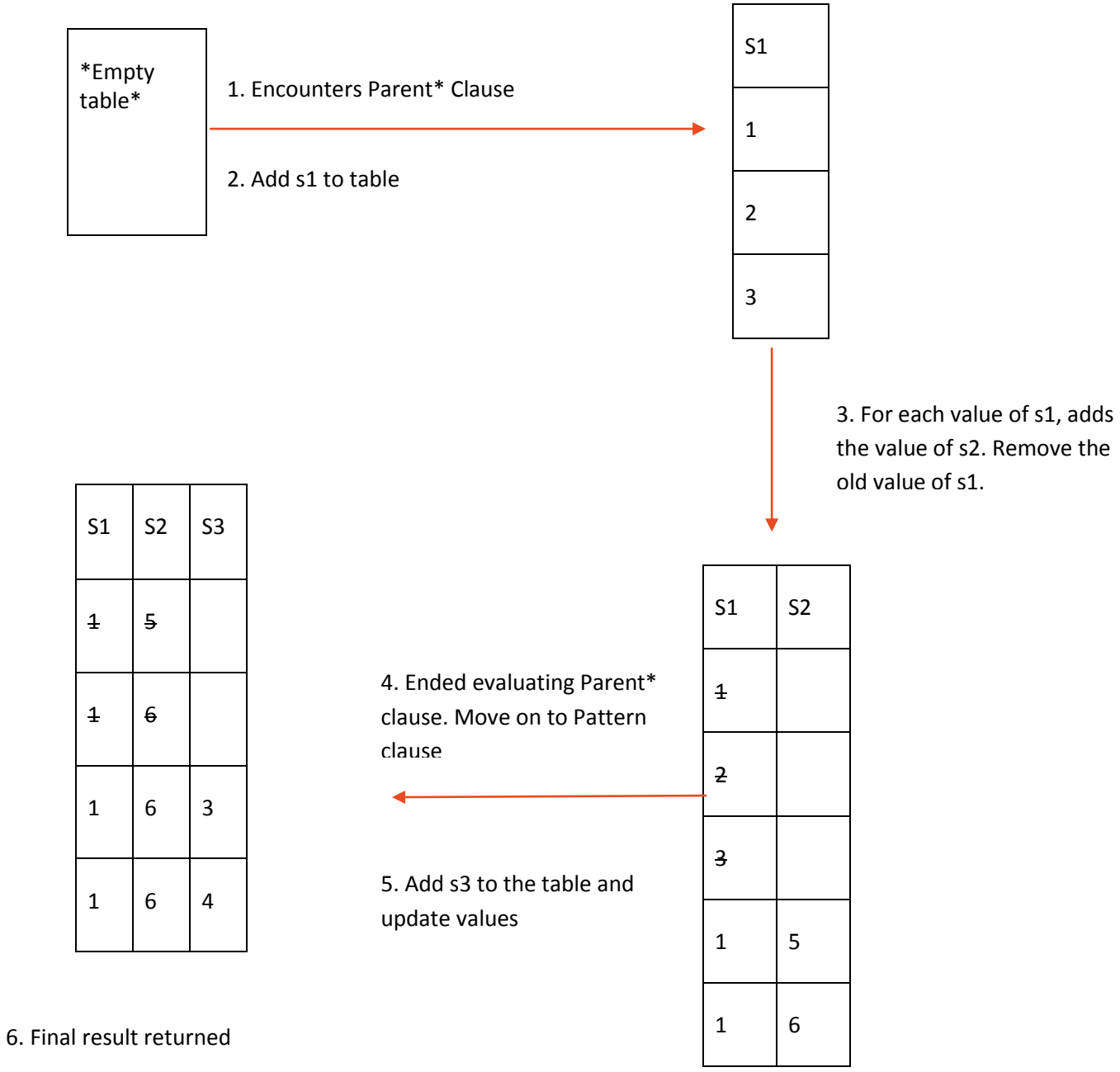
During the evaluation of a query, firstly, the QE will create an empty table for the result. After that, it will check all of the query's conditions one by one, attempting to update the list with values of parameters that satisfy the conditions. If it meets any condition that no value can satisfy, the QE will hang up the process and return an empty list as the result of the query.

After the process finishes, with all conditions being satisfied, the QE will take the result value and save it to the result list. This is the list that will be output to the user.

The sample query shown below will be used as an example to explain the query evaluation process

6.2.1. Manage the temporary results

Before evaluating a query, the QE creates an empty table. Keeping the above sample query in mind, the updating of this table is illustrated below.



During processing a query's condition, if a new value is found, the QE will update the list of values, by adding this new row. The QE will remove the old row and use the updated list to continue the evaluation. The list continues to be updated until there is no more condition left or a condition which cannot be satisfied appears.

In this way there is no need to consider merging results for the various clauses for evaluation. This updating of results table saves us from this hassle.

Finally at the end the result node will be read, from the query tree, to determine which result to return back to the QP.

6.2.2. Tuple Results

Based on the example shown above for managing the temporary results, it is quite obvious how the QE can also manage returning tuple results. In the event that the result node of the query tree points to returning more than one variable, the QE will just select the corresponding pairs of results from the final result table, as shown above.

6.2.3. Optimize the evaluation process

To better optimize our query evaluation, we have created a new class called Query Optimizer, QO. The job of this class is to reevaluate the resulting query tree, which has been built by the QPP. In this class 3 conditions are kept in mind to rearrange the query tree. This addition of the QO does not impact how the QE works. This shows good separation of concerns.

Let us now look at the 3 conditions in details.

i. Type of clause. The 3 types of clauses are such that, pattern and with. In this condition, we will arrange the query tree such that the clause with the smallest time complexity will be evaluated first. We have determined that the with clause takes the smallest time complexity to solve, followed by such that and pattern clauses. So the order of importance of clauses are:
    a. With
    b. Such that
    c. Pattern

ii. Number of symbols in each clause. It is obvious that the lesser the number of symbols in a clause, the easier it will be to solve. As such we for each clause, the number of symbols will be counted and smaller the number for that clause, we will put it at a higher priority to solve first. The thing to note:
    a. Num of symbols

iii. Symbol frequency. Throughout all of the clauses we will determine which symbol is used most frequently. If a clause contains a symbol that is highly used, we will solve that clause first, since it will make the solving of the other clauses easier. The thing to note is:
    a. Freq of each symbol

Now keeping in mind all of the above conditions, we will give each clause a ranking and rearrange the query tree according to this ranking.

Let's take a look at the query below,

stmt s1, s2, s3;

Select s1 such that Parent*(s1, s2) pattern s3(_, _"x+y*2"_) with s1.stmt#=10

Now after optimization, the query shown above would look something like this,

stmt s1, s2, s3;

Select s1 with s1.stmt#=10 such that Parent*(s1, s2) pattern s3(_, _"x+y*2"_)

7. Testing

7.1. Test Plan

In this iteration we continuously tested each and every component after changes were made. We made the decision that unit testing on each component would be done by the person implementing the component. Integration testing amongst components would be carried out by the people responsible for the specific components. Finally validation testing will be carried out by everyone at various points of progress during development.

In some aspects of the testing we have used assertions. Examples of where they were used will be shown in the figures shown below in the following sub-sections.

7.2. Examples of Test Cases

7.2.1. Unit Testing

7.2.1.1. PKB

Sample 1: Test MapTable

Test Purpose: To Test Map Table

Required Test Inputs: Map Table and CPPUnit test cases. An example is shown below

```
void MapTableTest::TestIsMappedStar() {
    int key1 = 10;
    int key2 = 20;
    int key3 = 30;
    int key4 = 40;

    MapTable<int> mapTable;
    mapTable.insert (key1, key2);
    mapTable.insert (key2, key1);
    mapTable.insert (key2, key3);

    CPPUNIT_ASSERT_EQUAL (true, mapTable.isMappedStar(key1, key3, true));
    CPPUNIT_ASSERT_EQUAL (false, mapTable.isMappedStar(key1, key4, true));
}

void MapTableTest::TestGetValues() {
    int key1 = 10;
    int key2 = 20;
    int key3 = 30;
    int key4 = 40;

    MapTable<int> mapTable;
    mapTable.insert (key1, key2);
    mapTable.insert (key2, key1);
    mapTable.insert (key2, key3);

    int expectedResultSize = 2;
    int actualResultSize = mapTable.getValues(key2).size();
    CPPUNIT_ASSERT_EQUAL (expectedResultSize, actualResultSize);
}

void MapTableTest::TestGetValuesStar() {
    int key1 = 10;
    int key2 = 20;
    int key3 = 30;
    int key4 = 40;

    MapTable<int> mapTable;
    mapTable.insert (key1, key2);
    mapTable.insert (key2, key1);
    mapTable.insert (key2, key3);
    mapTable.preCalculateStarTable();

    int expectedResultSize = 3;
    int actualResultSize = mapTable.getValuesStar(key1).size();
    CPPUNIT_ASSERT_EQUAL (expectedResultSize, actualResultSize);
}
```

Expected Test Results: “OK tests” if it passes all CPPUnit asserts

Other Requirements: None

Sample 2: Test List Table

Test Purpose: Test List Table

Required Test Inputs: List Table and CPPUnit test cases. An example is shown below.

```
CPPUNIT_TEST_SUITE_REGISTRATION( ListTableTest );

void ListTableTest::TestInsert() {
    ListTable <string> listTable;
    string element1 = "a";
    string element2 = "b";
    listTable.insert (element1);
    listTable.insert (element2);
    int expectedSize = 2;
    int actualSize = listTable.getSize();
    CPPUNIT_ASSERT_EQUAL (expectedSize, actualSize);
}

void ListTableTest::TestGetValue() {
    ListTable <string> listTable;
    string element1 = "a";
    string element2 = "b";
    listTable.insert (element1);
    listTable.insert (element2);

    CPPUNIT_ASSERT_EQUAL (listTable.getValue(0), element1);
    CPPUNIT_ASSERT_EQUAL (listTable.getValue(1), element2);
}

void ListTableTest::TestGetIndexes() {
    ListTable <string> listTable;
    string element1 = "a";
    string element2 = "b";
    string element3 = element1;
    listTable.insert (element1);
    listTable.insert (element2);
    listTable.insert (element3);

    int expectedResultSize = 2;
    int actualResultSize = listTable.getIndexes(element1).size();

    CPPUNIT_ASSERT_EQUAL (expectedResultSize, actualResultSize);
}
```

Expected Test Results: “OK tests” if it passes all CPPUnit asserts

Other Requirements: None

7.2.1.2. PQL

Sample 1: Test QP

Test Purpose: Test Query Processor

Required Test Inputs: Query Preprocessor and PQL Queries. An example is shown below.

```
void QueryPreprocessorTest::testBuildPatternCls() {
    QueryPreprocessor qp;
    string query = "assign a; Select a pattern a(_,_)";
    QueryRepresentator::reset();
    qp.Preprocess(query);

    Tree tree = QueryRepresentator::getQueryTree(0);
    TNode * root = tree.getRoot();
    // cout << endl << "TEST BUILD TREE: Query: assign a; Select a pattern a(_,_)" <<endl;
    // root -> printTNode();
}

void QueryPreprocessorTest::testBuildWithCls() {
    QueryPreprocessor qp;
    string query = "stmt s; prog_line p; Select BOOLEAN with p = 1";
    QueryRepresentator::reset();
    qp.Preprocess(query);

    Tree tree = QueryRepresentator::getQueryTree(0);
    TNode * root = tree.getRoot();
    /* cout << endl << "TEST BUILD TREE: Query: stmt s, prog_line p; Select BOOLEAN with p = 1" <<endl;
     root -> printTNode();*/
}

void QueryPreprocessorTest::testBuildTupleResult() {
    QueryPreprocessor qp;
    string query = "assign a1, a2; while w1, w2; Select <a1, a2>";
    QueryRepresentator::reset();
    qp.Preprocess(query);

    Tree tree = QueryRepresentator::getQueryTree(0);
    TNode * root = tree.getRoot();
    /*cout << endl << "TEST BUILD TREE: assign a1, a2; while w1, w2; Select <a1, a2>" <<endl;
    root -> printTNode();*/
}

void QueryPreprocessorTest::testBuildComplexQuery1() {
    QueryPreprocessor qp;
    string query = "assign a1, a2; while w1, w2; Select a2 pattern a1(\"x\",_) and a2(\"x\",_\"x\"_) such that Affects(
        a1, a2) and Parent*(w2, a2) and Parent*(w1, w2)";
    QueryRepresentator::reset();
    qp.Preprocess(query);

    Tree tree = QueryRepresentator::getQueryTree(0);
    TNode * root = tree.getRoot();
    /*cout << endl << "TEST BUILD TREE: Query: assign a1, a2; while w1, w2; Select a2 pattern a1(\"x\",_) and
    a2(\"x\",_\"x\"_) such that Affects(a1, a2) and Parent*(w2, a2) and Parent*(w1, w2)" <<endl;
    root -> printTNode();*/
}
```

Expected Test Results: “OK tests” if it passes all CPPUnit asserts

Other Requirements: None

Sample 2: Test QR

Test Purpose: Test Query Representator

Required Test Inputs: Query Representator and a pre-built QueryTree and SymbolTable. An example is shown below.

```
void QueryRepresentatorTest::testSymbolTable() {
    setUpData();

    CPPUNIT_ASSERT_EQUAL(3, QueryRepresentator::getSize());

    SymbolTable table1 = QueryRepresentator::getSymbolTable(0);
    SymbolTable table2 = QueryRepresentator::getSymbolTable(1);
    SymbolTable table3 = QueryRepresentator::getSymbolTable(2);

    CPPUNIT_ASSERT_EQUAL(1, table1.getSize());
    CPPUNIT_ASSERT_EQUAL(KEYWORD_STMT, table1.getType(0));
    CPPUNIT_ASSERT("s1"==table1.getName(0));

    CPPUNIT_ASSERT_EQUAL(2, table2.getSize());
    CPPUNIT_ASSERT_EQUAL(KEYWORD_STMT, table2.getType(0));
    CPPUNIT_ASSERT("s1"==table2.getName(0));
    CPPUNIT_ASSERT_EQUAL(KEYWORD_VAR, table2.getType(1));
    CPPUNIT_ASSERT("x"==table2.getName(1));

    CPPUNIT_ASSERT_EQUAL(2, table3.getSize());
    CPPUNIT_ASSERT_EQUAL(KEYWORD_STMT, table3.getType(0));
    CPPUNIT_ASSERT("s1"==table3.getName(0));
    CPPUNIT_ASSERT_EQUAL(KEYWORD_WHILE, table3.getType(1));
    CPPUNIT_ASSERT("w"==table3.getName(1));
}

void QueryRepresentatorTest::testQueryTree() {
    QueryTree tree1 = QueryRepresentator::getQueryTree(2);
    TNode root1 = *tree1.getRoot();

    CPPUNIT_ASSERT_EQUAL(1, root1.getNumChildren());
    /*cout<< endl;
    tree1.printTree();
    cout <<endl;*/
}
```

Expected Test Results: “OK tests” if it passes all CPPUnit asserts

Any other requirements: None

7.2.2. Integration Testing

7.2.2.1. Parser, PKB & Design Extractor

Test Purpose: Test the interaction of Paser and Design Extractor with PKB

Required Test Inputs: PKB, Parser and Design Extractor, and a simple source code.

```
void SystemTest2::testProcessModify() {
    string source = "..\\..\\Tests\\TestDesignExtractorSource.txt";
    Parser parser;
    PKB pkb;
    DesignExtractor designextractor;

    parser.parse(source);
    parser.buildPKB();
    designextractor.buildPKB();

    int A = pkb.getProcIndex("A").at(0);
    int B = pkb.getProcIndex("B").at(0);
    int C = pkb.getProcIndex("C").at(0);
    int D = pkb.getProcIndex("D").at(0);
    int E = pkb.getProcIndex("E").at(0);

    int x = pkb.getVarIndex("x");
    int a = pkb.getVarIndex("a");
    int c = pkb.getVarIndex("c");
    int k = pkb.getVarIndex("k");
    int loops = pkb.getVarIndex("loops");
    int j = pkb.getVarIndex("j");
    int b = pkb.getVarIndex("b");
    int fruit = pkb.getVarIndex("fruit");

    CPPUNIT_ASSERT(pkb.isCalls(C, D));
    CPPUNIT_ASSERT(pkb.isCalls(A, B));
    CPPUNIT_ASSERT(pkb.isCalls(C, E));
    CPPUNIT_ASSERT(pkb.isCalls(B, C));

    CPPUNIT_ASSERT(pkb.isCallStar(A, B));
    CPPUNIT_ASSERT(pkb.isCallStar(A, C));
    CPPUNIT_ASSERT(pkb.isCallStar(A, D));
    CPPUNIT_ASSERT(pkb.isCallStar(A, E));

    vector<int> modifiedvars = pkb.getModifiedVarAtProc(A);
    sort(modifiedvars.begin(), modifiedvars.end());
    int vars[] = {x, a, c, k, loops, j, b, fruit};
    vector<int> expectedvars(vars, vars + sizeof(vars) / sizeof(int) );
    sort(expectedvars.begin(), expectedvars.end());

    CPPUNIT_ASSERT_EQUAL(expectedvars.at(0), modifiedvars.at(0));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(1), modifiedvars.at(1));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(2), modifiedvars.at(2));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(3), modifiedvars.at(3));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(4), modifiedvars.at(4));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(5), modifiedvars.at(5));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(6), modifiedvars.at(6));
    CPPUNIT_ASSERT_EQUAL(expectedvars.at(7), modifiedvars.at(7));
```

```
procedure A {
    call B;                         \\1
    x = 3;                          \\2
    k = x + w;                      \\3
}

procedure B {
    call C;                         \\4
    a = b + c;                      \\5
    c = new;                        \\6
}

procedure C {
    call D;                         \\7
    call E;                         \\8
    loops = yes;                    \\9
    j = k;                          \\10
}

procedure D {
    b = c + a;                      \\11
}

procedure E {
    fruit = loops;                  \\12
}
```

Expected Test Results: “OK tests” if it passes all CPPUnit asserts

Other Requirements: None

7.2.3. Validation Testing

Sample 1

Test Purpose: Test Affects and AffectsStar Relationship

Required Test Inputs: The whole system. Given input is source code and query file.

```
procedure Alpha {
   idx = 1;
   y = idx;
   x = idx *y + 2;
   call Charlie;
   t = idx + y;
   call Delta;
   t = idx + y;
   if idx then {
      k = y - x;
      while z {
           x = x + idx;
        z = x + 1; }
      y = x + idx; }
   else {
      y = x + idx;
      call Delta;
      z = z - 1; }
   call Bravo;
   if x then {
      a = 5;
      b = z;
      while x {
        c = idx + a + 5;
        if c then {
           idx=a;
           a = b;
           b = c;
           c = idx;
           idx = z;
           z = a;
        }
        else {
           call Bravo;
           b = idx;
           c = b;
           a = z;
           z = c;
           c = idx;
        }
      }
    }
    else {
       call Charlie;
       if z then {
          while a{
         call Iftest;
      }
      y = x+ idx +a+b+c+k;
       }
       else{
      call Delta;
       }
    }
 }

procedure Bravo {
 while z {
 idx = x + 3 * y + z;
 call Charlie;
 z = z - 1; }
 x = idx; }
```

Expected Test Results: As shown in figure below.

```
1 - Affects ::
assign a1;
Select a1 such that Affects(103, a1)
103, 104, 113, 114, 119
5000
2 - Affects ::
assign a1;
Select a1 such that Affects(104, a1)
103, 113
5000
3 - Affects ::
assign a1;
Select a1 such that Affects(105, a1)
103, 104, 118
5000
4 - Affects ::
assign a1;
Select a1 such that Affects(106, a1)
none
5000
5 - Affects ::
assign a1;
Select a1 such that Affects(107, a1)
103, 107
5000
6 - Affects ::
assign a1;
Select a1 such that Affects(108, a1)
103, 108
5000
7 - Affects ::
assign a1;
Select a1 such that Affects(109, a1)
109
5000
8 - Affects ::
assign a1;
Select a1 such that Affects(113, a1)
114, 115, 116
5000
9 - Affects ::
assign a1;
Select a1 such that Affects(114, a1)
103, 113, 114, 115, 119
5000
10 - Affects ::
assign a1;
Select a1 such that Affects(115, a1)
103, 104, 116
5000
```

Other Requirements: None

Sample 2

Test Purpose: Test returning Tuple Results

Required Test Inputs: The whole system, given input is source code and query file.

```
procedure Example { x = 2;
  z = 25;
  i = 13;
  while i {
    x = x - 1;
    if x then {
      z = x + 1; }
    else { y = z + x; }
    z = z + x + i;
    call q;
    i = i - 1; }
  call p; }
procedure p { if x then {
    while i {
      x = z * 3 + 2 * y;
      call q;
      i = i - 1; }
    x = x + 1;
    z = x + z; }
  else { z = 10; }
  z = z + x + i; }
procedure q { if x then {
    z = x + 1; }
  else { x = z + x; } }
```

Expected Test Results: As shown in in figure below.

```
1 - MultipleClauses, Typed :: Modifies and Uses
variable v; stmt s;
Select <s,v> such that Modifies("Example",v) and Uses(s,v) and Modifies(s,v)
4 i, 4 x, 4 z, 5 x, 6 z, 9 z, 10 z, 10 x, 11 i, 12 x, 12 i, 12 z, 13 x, 13 i, 13 z, 14 i, 14 x, 14 z, 16 x, 16 z,
17 i, 18 x, 19 z, 21 z, 22 x, 22 z, 24 x
5000
2 - MultipleClauses, Typed :: Next* and Parent*
if i; assign a;
Select <i,a> such that Parent*(i,a) and Next*(a, 20)
none
5000
3 - MultipleClauses, Typed :: Parent*
stmt s1, s2, s3;
Select <s1,s2> such that Parent*(s1,s2) and Parent*(s2, s3)
4 6, 13 14
5000
4 - MultipleClauses, Typed :: Calls and Modifies
procedure p1,p2; variable v;
Select <p1,v> such that Modifies(p1,"x") and Calls(p1,p2)
Example x, Example y, Example i, Example z, p x, p y, p i, p z
5000
5 - MultipleClauses, Typed :: Calls and Modifies
procedure p1,p2;
Select <p1,p2> such that Modifies(p1,"x") and Calls(p1,p2)
Example p, Example q, p q
5000
6 - MultipleClauses, Typed :: Next and Calls
procedure p; prog_line n;
Select <n,p> such that Next(n,6) and Calls(p,"q")
5 Example, 5 p
5000
7 - MultipleClauses, Typed :: Next and Calls
procedure p; prog_line n; if i;
Select <n,i> such that Next(n,6) and Calls(p,"q")
5 6, 5 13, 5 22
5000
8 - MultipleClauses, Typed :: Next* and Next
prog_line n1,n2; stmt s;
Select <n1,n2> such that Next(n2,n1) and Next(s,1)
none
5000
9 - MultipleClauses, Typed :: Parent and Modifies
variable v; stmt s1;
Select <v, s1> such that Parent(s1,8) and Modifies(s1,v)
y 6, z 6
```

Other Requirements: None

8. Discussion

In this iteration, the biggest challenge that we faced was implementing the Next* and Affects/Affects* relation correctly. Since we had not implemented the DE in iteration 2, we did not cover all of the possible cases covered for Next* and thus for Affects as well. We realized that we missed out a major chunk of relationships between Modifies and Uses for procedure calls and Modifies/ Uses statements. Our tutor helped us pin-point this issues during our consultations. Since we added the DE, we have managed to implement the Next and Affects relationships correctly and cover all of the cases.