



NUS
National University
of Singapore

CS3201/2 Software Engineering Project

Iteration 3 Report

Filbert Cheong	A0121261Y	filbert.cheong@gmail.com	84286730
Lim Jie	A0139221N	limjie1994@gmail.com	97109212
Lim Wei Jie	A0139128A	limweijie1994@gmail.com	88766462
Niveetha Nair	A0126090N	niveetha.nair@gmail.com	92271064

Content

CS3201/2 Software Engineering Project	0
Iteration 3 Report	0
Content	1
Abstract	4
1 Scope	5
Iteration 1	5
Iteration 2	5
Iteration 3	5
2 Development Plan	6
2.2 Component Developers	6
2.2 Component Implementations	7
3 SPA Design and Components	9
Figure 3.1 SPA Architecture	9
Front-end	10
Query Processor (Back-end)	10
3.1 SPA Components	10
3.1.1 Parser	10
High-Level Procedure	11
Statement Match / Validation	13
Parser Variables	13
Low-Level Procedure	14
Utility methods	19
3.1.2 Program Knowledge Base (PKB)	20
Procedure / Variable Index Table	20
Populating Relationship Tables	21
Follow Class	21
Parent Class	23
Modify Class	26

Call Class	28
Next Class	29
Non-precomputed Relationships	30
Next* Class	30
Affects Class	30
Affects* Class	31
Design Extractor	31
3.1.3 Query Preprocessor	31
RelationshipTable	32
SynonymEntityList	32
QueryElement	33
QueryStatement	34
QueryValidator	35
3.1.4 Query Evaluator	39
Query Evaluator Architecture	39
queryAnalyzer	41
queryOptimizer	42
Intermediate Table Result - MergedQueryTable	42
Hash-join (merging) algorithm	43
Query Evaluator Process	45
3.2 Design Decision	52
3.2.1 Parser	52
Storing the Parent	53
Storing the Nesting Level	53
3.2.2 PKB	54
Creating variable and procedure index tables	54
Storing of relationships in PKB	55
3.2.3 Query Pre-processor	56
Changing QueryTree to QueryStatement object	56
RelationshipTable & withClauseTypeBank	56

3.2.4 Evaluator	57
3.3 Component Interactions	58
4 Documentation and Coding Standards	59
5 Testing	60
5.1 Unit Testing	60
5.1.1 Parser	60
Parser Unit Test - Assign Statement Validation	60
5.1.2 PKB	61
PKB Unit Test - Modifies Updates Containers	61
PKB Unit Test - Parent Set Correctly	62
5.1.3 Preprocessor	62
Preprocessor Unit Test - Correct Creation of queryStatement Object	62
5.1.4 Evaluator	63
Evaluator Unit Test - Check Intermediate Tables Merging	63
5.2 Integration Testing	64
5.2.1 Front-End Integration Test	64
Front-End Test - Integration between Parser and PKB	64
5.2.2 Query Preprocessor Test	65
Query Validation - testQueryPattern	65
Query Validation - testQueryAll	65
5.3 System (Validation) Testing	66
5.3.1 Source 6	66
Table 5.3.1.1	70
5.3.2 Source 7	70
Table 5.3.2.1	71
5.3.3 Source 8	72
Table 5.3.3.1	73
5.3.4 Regression Testing	73
5.3.5 AutoTester	73
Figure 5.3.4.1	74

Figure 5.3.4.2	74
6 Discussion	75
7 Documentation of Abstract APIs	75
7.1 PKB	75
7.2 Preprocessor	77
7.2.1 SynonymEntityPair	77
7.2.2 QueryValidator	77
7.2.3 Relationship	79
7.2.4 RelationshipTable	79
7.2.5 QueryElement	79
7.2.6 QueryStatement	81
7.3 Evaluator	82
7.3.1 queryAnalyzer	82

Abstract

The Static Program Analyzer (SPA) is a program that takes in code written in the SIMPLE language, and outputs results to queries regarding the input; example queries would be “What statements modifies the variable “x”, or what statements are executed after statement 27?” This report documents the production of such a system, and details the components, their interactions, decision decisions and testing methods during the development process.

1 Scope

The scope of our project has been sectioned by iterations; they are as follows,

Iteration 1

Iteration I focused on creating a functional static program analyzer (SPA) for programs written in the SIMPLE language. Our SPA can parse in a single procedure, containing assignment, if-else-then and while statements. The parser is able to call upon API to populate relationship tables in the program knowledge base (PKB).

Our SPA is able to process queries containing Follows, Follows*, Parent, Parent*, Modifies and Uses, and also with at most one “pattern” and “such that” clauses. Once pre-processing is complete, our SPA passes it on to the evaluator to obtain and display the results.

Iteration 2

For Iteration II, the PKB focused on the implementation of Calls and Calls*. The parser is now able to take in zero or more spaces, and zero or more tabs (according to SIMPLE standard); it is also able to take in empty lines and close brackets at the start of parsing. Lastly, the parser can handle Call statements, and call the relevant APIs to populate relationship tables in the PKB.

As for the SPA back-end, the preprocessor can now handle multiple relationships and “pattern” clauses; it can now also accept “with” clauses and handle BOOLEAN select. The evaluator can respond to multiple “with” and “pattern” clauses, multiple common synonyms, and BOOLEAN select.

Iteration 3

For the third and final iteration, the PKB has implement focused on the implementation of Next, Next*, Affects, Affects*. However, due to time constraint and lack of manpower as we are only a team of 4 people, the ‘Affects*’ clause may not be implemented to perfection.

As for the SPA-back-end: Tuples, Next, Next*,Affects, Affects* is also implemented at both the query preprocessing and evaluator side. Furthermore, optimization has been done to evaluate queries at a faster rate.

2 Development Plan

2.2 Component Developers

Iteration 1					
	Filbert	Limjie	Niveetha	Weijie	Vivienne
Parser		*			*
PKB		*	*		
Preprocessor				*	
Evaluator	*				
Testing	*	*	*	*	
Documentation	*	*	*	*	

Iteration 2				
	Filbert	Limjie	Niveetha	Weijie
Parser		*		
PKB			*	
Preprocessor				*
Evaluator	*			
Testing	*	*	*	*
Documentation	*	*	*	*

Iteration 3				
	Filbert	Limjie	Niveetha	Weijie
Parser		*		
PKB			*	
Preprocessor				*
Evaluator	*			
Testing	*	*	*	*
Documentation	*	*	*	*

2.2 Component Implementations

Parser	<ul style="list-style-type: none"> • Validation of SIMPLE code • Parsing of SIMPLE code into extractable information • Update PKB using API • One-time parse • One-time loop of CFG • Split according to statement type
PKB	<ul style="list-style-type: none"> • Follows / Follows* • Parent / Parent* • Modifies / Procedure Modifies • Uses / Procedure Modifies • Procedure Calls / Statement Calls / Calls* • Next / Next* • Affects / Affects* • Pattern Recognition
Preprocessor	<ul style="list-style-type: none"> • Validation of “such that” clauses (Follows/Follows*, Parent/Parent*, Modifies, Uses, Next/Next*, Calls/Calls*, Affects/Affects*) • Validation of “pattern” clauses (assign, if-else-then, while) • Validation of “with” clauses • Validation of multiple clauses of combination “such-that, pattern

	<p>or with” clauses</p> <ul style="list-style-type: none"> • Validation of “tuples/synonym/attrName/BOOLEAN”
Evaluator	<ul style="list-style-type: none"> • Query PKB for “such that” clauses (Follows / Follows*, Parent / Parent*, Modifies / Procedure Modifies, Uses / Procedure Uses, Next / Next*, Affects / Affects*) • Query PKB for “pattern” clauses (assign, if-else-then, while) • Query PKB for “with” clauses • Join on common synonyms • Restrict clause results based on synonym entity design • Evaluating multiple clauses comprised of multiple common synonym • Evaluating multiple clauses in consideration of “tuples/synonym/attrName/BOOLEAN” • Optimization of clauses

3 SPA Design and Components

The Static Program Analysers consists of 4 major components: parser, program knowledge base (PKB), query pre-processor, and finally the query evaluator. The figure below displays the high-level design of our SPA,

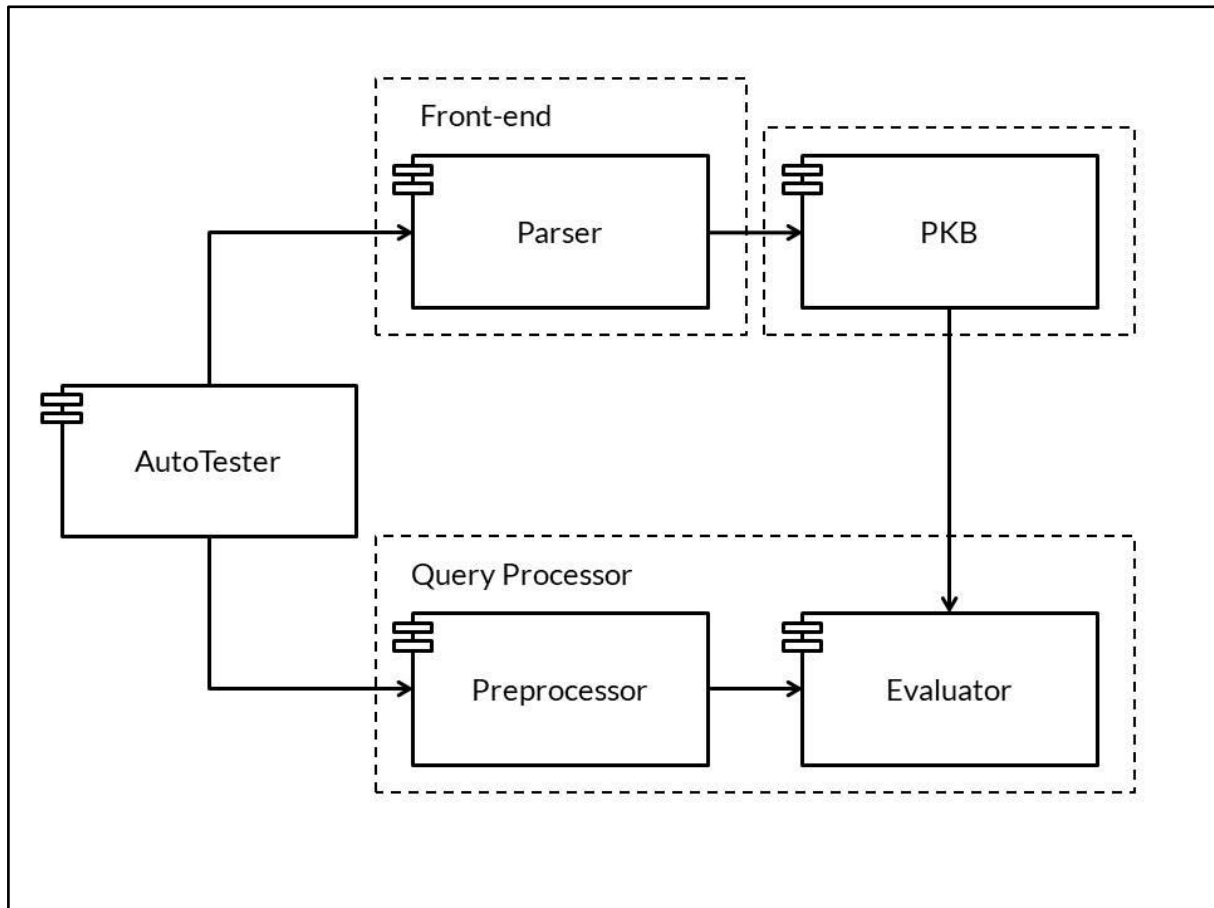


Figure 3.1 SPA Architecture

Front-end

- Parser: Parse and validate SIMPLE code. Calls relevant PKB APIs and populates relationship tables in the PKB.

PKB: Stores all relationship information of parsed in code. Contains APIs to populate its relationship tables and get its values.

Query Processor (Back-end)

- Preprocessor: Validates queries and creates queryStatement object to be passed to the evaluator
- Evaluator: Calls PKB APIs to respond to validated queries and returns result.

3.1 SPA Components

3.1.1 Parser

The parser validates and parses SIMPLE source codes, calling relevant PKB APIs to populate its relationship tables; the parser populates Variable, Procedure, Follows / Follows*, Parent / Parents*, Uses / Procedure Uses, Modifies / Procedure Modifies, Calls / Calls*, Next and statement type (assign / if-else-then / while / call) tables. At the end of parsing, the parser calls the PKB method createCFG to populate the next table. All information is stored bi-directional (e.g. statement 1 modifies “x” is also stored as “x” is modified by statement 1) to allow for faster and easier retrieval of results to queries which require the reverse of how the data is stored (e.g. getModifies(s, “x”) can be answered using getModifiedBy(“x”).

High-Level Procedure

- The input SIMPLE source code is read in line-by-line, splitting the line by spaces and looking for a statement type match; if no match is found, the code terminates.
- Validation of source code is done through statement match. When a match is found, corresponding PKB methods are called to store the relationships in the SIMPLE source code.

Validation and matching of statements	PKB API calls
For empty lines,	
For else lines, check for the keyword else followed by an open bracket	insertElse - stores the location of else statements
For procedure lines, check for keyword procedure, a valid name (by SIMPLE grammar rules) and an open bracket	insertStatementList - stores the location of the start of a statementList setLastline - stores the location of the end of procedure
For lines with close brackets, check for open bracket at the start, and the else keyword (optional)	if else is present, insertElse - stores the location of else statements
For code lines with statement numbers (above code lines have no statement numbers),	For all statements, setFollows - stores the follow relationship, if they follow a statement setParent - stores the parent relationship, if they have a parent statement setProcedure - stores procedure the statement belongs to
For assign statements, find the equal sign "=", and ensure variable on the left has a valid name, and expression on the right is well formed with valid variable names	setStatementType - stores the statement type (i.e. call type, while type, etc.) setModifies - stores the variable the statement modifies setProcModifies - stores the variable the procedure modifies addConstant - stores the constants the

	<p>SIMPLE code contains</p> <p>setUses - stores the variable the statement uses</p> <p>setProcUses - stores the variable the procedure uses</p> <p>addAssignPattern - stores the expression on the right of equal (for pattern matching)</p>
For if statements, check for the keywords if and then, with a valid name in between, followed by an open bracket	<p>setStatementType - stores the statement type</p> <p>addIfPattern - stores the control variable for pattern matching</p> <p>setUses - stores the control variable the statement uses</p> <p>setProcUses - stores the control variable the procedure uses</p>
For while statements, check for the keyword while, followed by a valid name and close bracket	<p>setStatementType - stores the statement type</p> <p>addWhilePattern - stores the control variable for pattern matching</p> <p>setUses - stores the control variable the statement uses</p> <p>setProcUses - stores the control variable the procedure uses</p>
For call statements, check for the keyword call, followed by a valid name and close bracket	<p>setStatementType - stores the statement type</p> <p>setCalls - stores the call statement, this includes statement number, the procedure calling and the procedure being called</p>
For anything else,	terminate

CFG Creation

- createCFG - algorithm uses recursion to populate the next table statement by statement, not line by line

- With a single parse and one run of createCFG, we are able to populate all relationship tables contained in the PKB. All statements are parsed regardless of the amount of spaces between them; statements must follow SIMPLE grammar standards.
- Parsing terminates when a line finds no match, when the number of open and close brackets are not the same at the end, when there is a variable or procedure with an invalid name, or if there is an empty container statement list.

Statement Match / Validation

Statement match / validation is done by (1) checking for keywords, and (2) names are checked if they are valid.

Keyword Check	<ul style="list-style-type: none"> • Procedure : 'procedure' procedure_name '{' • While : 'while' variable_name '{' • If : 'if' variable_name 'then' '{' • Else : 'else' '{' • assign variable_name '=' expr ';' • Empty line : check that line is empty (i.e. nothing on line) • Call : 'call' procedure_name ';' • assign call : check for close brackets at the end (i.e. nothing except close brackets)
Name Check	scan the name character by character, ensuring the first character is a letter, and the following characters are numbers or letters

Parser Variables

- The parser matches source code lines to a statement type: procedure, assign, while, if, call, empty line, close bracket, open bracket (line with open brackets with / without else) and else.
- The parser contains (1) an int lineCounter, which tracks the number of statements and populates the PKB with the relevant information, (2) a vector<int> Parent stack, to help keep track of the immediate parent of the currently-parsing statement line. and (3) procName, to keep track of the current procedure.
- When a line is read, lineCounter remains unchanged and the next line is read; it only increases when a statement is read.

- Additional variable, and boolean variables (1) isSameLevel and (2) isNewContainer, helps with the parsing of statements in containers. For example, when an else statement is read, the if statement is pushed onto the Parent stack. Variable (3) prevFollow keeps track of the previous statement to follow when exiting a container. As we do so, isSameLevel is set to false, and isNewContainer is set to true; lineCounter is left unchanged.

Low-Level Procedure

At a procedure line,	<ol style="list-style-type: none"> (1) nestLevel and the size of Parent stack is checked to see if they are 0 - this is to ensure the number of open and close brackets before this line match-up (2) if current line is not the first line of the source code, it will be stored in the PKB as the last line of the previous procedure (3) Procedure is not a statement, hence lineCounter remains unchanged (4) procedure_name is stored in the PKB (5) a dummy variable is pushed onto the Parent stack to indicate a new statement list, and nestLevel is increased (6) isNewContainer and toSetFirstLine is set to true, and isSameLevel is set to false
At lines starting with close brackets,	<ol style="list-style-type: none"> (1) it could be one of two scenarios, <ul style="list-style-type: none"> • there is an else at the end, or • there is nothing at then end (2) loop through close brackets, decrease nextLevel accordingly, and pop Parent stack until current line is follows Parent on stack (3) isSameLevel is set to false to ensure the popped value is set to the previous follow, and not the immediate previous line; this is used for the case when exiting a container, and the previous line is the while / if statement (not the immediate previous line) (4) if during this process an else statement is found, we push back the if statement back into the Parent stack, as statements in the else container have the same parent
At lines with statement	<ol style="list-style-type: none"> (1) if isNewContainer is false, and the current line is of the same level (check using isSameLevel) as the previous line,

numbers,	<p>setFollows(lineCounter - 1, lineCounter); if isSameLevel is false, setFollows(prevFollow, lineCounter)</p> <p>(2) if Parent.size() is more than one, setParent must be called, using the Parent stack, setParent(lineCounter, line on Parent stack)</p>
At an assign line,	<p>(1) set statement type of line to be assign in PKB</p> <p>(2) as variables of the left of '=' are modified, the relevant Modify APIs (from the PKB) are called - setModifies(statement number, current line, variable) and setProcModifies(statement number, procedure name, variable)</p> <p>(3) Constants in assign statements are stored in the PKB using the addConstant function</p> <p>(4) as variables of the right of '=' are used, the relevant Use APIs (from the PKB) are called - setUses(statement number, current line, variable) and setProcUses(statement number, procedure name, variable)</p> <p>(5) the expression will be stored in the PKB using the addPattern function</p> <p>(6) isNewContainer is set to false, and isSameLevel is set to true</p>
At a while / if line,	<p>(1) statement type of current line is set and stored in the PKB</p> <p>(2) pattern is stored in the PKB by calling the corresponding PKB APIs - for if statements: addIfPattern, and for while statements: addWhilePattern</p> <p>(3) APIs setUses and setProcUses are called to store the Uses relationship - setUses(statement number, current line, variable) and setProcUses(statement number, procedure name, variable)</p> <p>(4) as while / if are the start of container statements, the line is pushed onto the Parent stack, nestLevel is increased, isSameLevel is set to false, and isNewContainer is set to true</p>
At a call line,	<p>(1) statement type of current line is set and stored in the PKB</p> <p>(2) corresponding API is called to store the statement number, procedure the statement is contained in, and procedure the statement is calling - setCalls(statement number, procedure calling, procedure being called)</p> <p>(3) isSameLevel is set to true, and isNewContainer is set to false</p>

If close bracket
exists at end of
line,

follow procedures for when line starts with close brackets

Example Code (aid visualisation of parser variables and PKB population)

Source Code	Variable change and API calls
procedure first {	insertStatementList(1); procName = first; Parent.push_back(0); isNewContainer = true; isSameLevel = false;
x = y + 5;	lineCounter++; setStatementType(1, ASSIGN); setModifies(1, x); setProcModifies(first, x); setUses(1, y); setProcUses(first, y); addConstant(5); addAssignPattern(1, x, y+5); isNewContainer = false; isSameLevel = true;
while x {	lineCounter++; setFollows(1, 2); addWhilePattern(2, x); setStatementType(2, WHILE); setUses(2, x); setProcUses(first, x); Parent.push_back(2); isSameLevel = true; isNewContainer = false;
y = z;	lineCounter++; setParent(2, 3); setStatementType(3, ASSIGN);

	setModifies(3, y); setProcModifies(first, y); setUses(3, z); setProcUses(first, z); addAssignPattern(3, y, z); isNewContainer = false; isSameLevel = true;
}	isSameLevel = false; prevFollow = Parent.back(); Parent.pop_back();
if z then {	setFollows(2, 4); setStatementType(4, IF); addIfPattern(4, z); setUses(4, z); setProcUses(first, z); Parent.push_back(4); isSameLevel = false; isNewContainer = true;
a = b * c;	setParent(4, 5); setStatementType(5, ASSIGN); setModifies(5, a); setProcModifies(first, a); setUses(5, b); setProcUses(first, b); setUses(5, c); setProcUses(first, c); addAssignPattern(5, a, b*c); isNewContainer = false; isSameLevel = true;
}	isSameLevel = false; prevFollow = Parent.back(); Parent.pop_back();
else {	insertElse(6); Parent.push_back(4); isSameLeve = false;

	isNewContainer = true;
a = 2;	setParent(4, 6); setStatementType(6, ASSIGN); setModifies(6, a); setProcModifies(first, a); addConstant(2); addAssignPattern(6, a, 2); isNewContainer = false; isSameLevel = true;
}	isSameLevel = false; prevFollow = Parent.back(); Parent.pop_back();
}	isSameLevel = false; prevFollow = Parent.back(); Parent.pop_back();
procedure second {	insertStatementList(7); setLastline(first, 6); procName = second; Parent.push_back(0); isNewContainter = true; isSameLevel = false;
call first;	setStatementType(7, CALL); setCalls(7, second, first); isSameLevel = true; isNewContainer = false; setCalls will find out which variables "first" uses and modifies, and set 7 and procedure "second" to use / modify it
}	isSameLevel = false; prevFollow = Parent.back(); Parent.pop_back();

Utility methods

A utility method was created to contain functions that facilitated trimming of white spaces, splitting a string based on a delimiter, changing an expression into a vector of its variables and constant, removing the operators, etc. As these functionalities were used by both the parser and PKB, and sometimes the preprocessor, it was created separate from the four major components.

3.1.2 Program Knowledge Base (PKB)

The Program Knowledge Base (PKB) contains API that sets and retrieves relationships in SIMPLE programs parsed through our SPA.

Procedure / Variable Index Table

Every procedure / variable name parsed into our SPA is indexed for easy referencing and storing of relationships. This index is private to the PKB class (i.e. parser, preprocessor and evaluator sets / retrieves procedure / variable names in its original string form).

The PKB contains functions `getVarIndex` / `getProcIndex` that returns the index of a string variable already in the index table, and if non-existent, creates one and returns the index. This “index” is the location number of the string in the `vector<string>` index table.

PKB
<ul style="list-style-type: none">- <code>varIndexTable : vector<string></code>- <code>procIndexTable : vector<string></code>
<ul style="list-style-type: none">- <code>getVarIndex() : int</code>- <code>getProcIndex : int</code>

UML diagram of indexing in the PKB

Populating Relationship Tables

As the parser takes in the input SIMPLE program, it calls upon PKB's relationship table populating APIs. The PKB creates a single object of each relationship type to access its setter functions. Relationship tables, though setting and getting are done in the PKB, are contained in each respective class (i.e. all Follows relationship data are stored in the Follows class). In other words, the PKB is an interface for the parser and evaluator to set and get, respectively, the relationships of the source program.

PKB
<ul style="list-style-type: none">- follow : Follow- parent : Parent- modify : Modify- use : Use- call : Call
<ul style="list-style-type: none">+ setFollows(int statementNum1, int statementNum2) : void+ setParent(int statementNum1, int statementNum2) : void+ setModifies(int statementNum, string varName) : void+ setProcModifies(string procName, string varName) : void+ setUses(int statementNum, string varName) : void+ setProcUses(string procName, string varName)+ setCalls(int statementNum, string procName1, string procName2) : void

UML diagram of relationship setter APIs in the PKB

The relationship classes were designed to limit the number of setters the parser needs to call when storing the relationship of the program (e.g. when PKB setFollows(1, 2), the function implicitly calls setFollowedBy(2, 1)).

Follow Class

Follow relationships are between statement numbers. The class contains four tables - (1) follow, (2) followedBy, (3) followStar, and (4) followedByStar. Bi-directional storage of values allows responding to queries such as Follows(s1, 2) easily; we call getFollowedBy(2).

The function `setFollow` implicitly calls `setFollowedBy`, `setFollowsStar` and `setFollowedByStar`. `setFollowStar` is set by looping through the `followedByTable` of the first argument and calling `setFollow` during this loop; `setFollowedByStar` is set here as well.

Follow	
-	<code>followTable : vector<int></code>
-	<code>followedByTable : vector<int></code>
-	<code>followsStarTable : vector<vector<int>></code>
-	<code>followedByStarTable : vector<vector<int>></code>
+	<code>setFollow(int s1, int s2) : void</code>
-	<code>setFollowedBy(int s1, int s2) : void</code>
-	<code>setFollowsStar(int s1, int s2) : void</code>
-	<code>setFollowedByStar(int s1, int s2) : void</code>

UML diagram of setter functions in Follow class

Example Code (aid visualisation of Follow class table population)

- (1) At line 1, `setFollow(1, 2)` is called; this populates the `followTable`, and implicitly populates the `followedBy`, `followStar` and `followedByStarTable`. `followStar` checks index 1 of the `followedByTable`, at this stage it is empty.

```

procedure first{
  x = y + 1;
  y = z + 2;
  z = 3;
}

```

followTable	
Index	Stmt Num
0	
1	2

followedByTable	
Index	Stmt Num
0	
1	
2	1

followStarTable	
Index	Stmt Num
0	
1	2

followedByStarTable	
Index	Stmt Num
0	
1	
2	1

- (2) At line 2, `setFollow(2, 3)` is called. Population of tables follows (1), except at this point, `getFollowedBy(2)` will return statement 1 - here, `setFollowStar(1, 3)` and `setFollowedByStar(3, 1)` is called.

<pre> procedure first{ x = y + 1; y = z + 2; z = 3; } </pre>	followTable		followedByTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2	1	
	2	3	2	1
			3	2
	followStarTable		followedByStarTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2, 3	1	
	2	3	2	1
			3	2, 1

Follow class also throws an invalid argument if the statement number is less than or equal to zero, or if the first statement number if the second argument in setFollow(arg1, arg2) is smaller than the first.

Parent Class

Parent relationships, like Follow, are between statement numbers. The class contains four tables - (1) parent, (2) child, (3) parentStar, and (4) childStar. Again, bi-directional storage of values allows responding to queries such as Parent(2, s1) easily; we call getChild(2).

The function setParent implicitly calls setChild, setParentStar and setChildStar. setParentStar is set by looping through the parentTable of the first argument and calling setParent during this loop; setChildStar is set here as well.

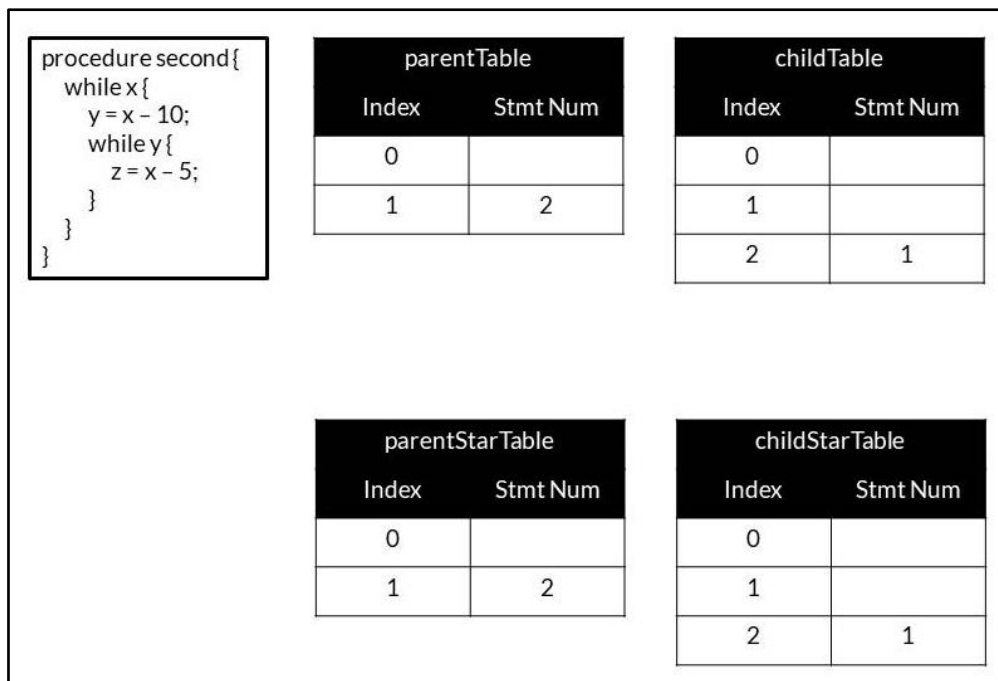
Parent	
-	parentTable : vector<int>
-	childTable : vector<int>
-	parentStarTable : vector<vector<int>>
-	childStarTable : vector<vector<int>>
+	setParent(int s1, int s2) : void

- setChild(int s1, int s2) : void
- setParentStar(int s1, int s2) : void
- setChildStar(int s1, int s2) : void

UML diagram of setter functions in Parent class

Example Code (aid visualisation of Parent class table population)

- (1) Skipping forward to line 2, setParent(1, 2) is called; this populates the parentTable, and implicitly populates the childTable (setChild(2, 1)), parentStarTable (setParentStar(1, 2)) and childStarTable (setChildStar(2, 1)). parentStar checks index 1 of the parentTable, at this stage it is empty.



- (2) Setting of Parent relationships at line 3 is similar to that of (1).

<pre> procedure second{ while x{ y = x - 10; while y{ z = x - 5; } } } </pre>	parentTable		childTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2,3	1	
	parentStarTable		childStarTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2,3	1	
			2	1
			3	1

(3) At line 4, setParent(3, 4), and implicitly setChild(4, 3), are called. Here, getParent(3) will return 1; setParentStar(1, 4) and setChildStar(4, 1) are called.

<pre> procedure second{ while x{ y = x - 10; while y{ z = x - 5; } } } </pre>	parentTable		childTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2,3	1	
	parentStarTable		childStarTable	
	Index	Stmt Num	Index	Stmt Num
	0		0	
	1	2,3,4	1	
			2	1
			3	1
			4	3,1

Parent class, like Follow class, throws an invalid argument if the statement number is less than or equal to zero, or if the first statement number if the second argument in setParent(arg1, arg2) is smaller than the first.

Modify Class

Modify class, unlike Follow and Parent, stores relationships between (1) statement number and variable, and (2) procedure and variable. The class contains four tables - (1) modifies, (2) modifiedBy, (3) procModifies, and (4) procModifiedBy. Bi-directional storage of these relationships allows easy response to queries such as Modify(s1, "x") easily; we call getModifiedBy("x").

Modify
<ul style="list-style-type: none">- modifiesTable : vector<set<int>>- modifiedByTable : vector<set<int>>- procModifiesTable : vector<set<int>>- procModifiedByTable : vector<set<int>>
<ul style="list-style-type: none">+ setModifies(int statementNum, int varName) : void+ setProcModifies(int procName, int varName) : void- setModifiedBy(int varName, int statementNum) : void- setProcModifiedBy(int varName, int procName) : void

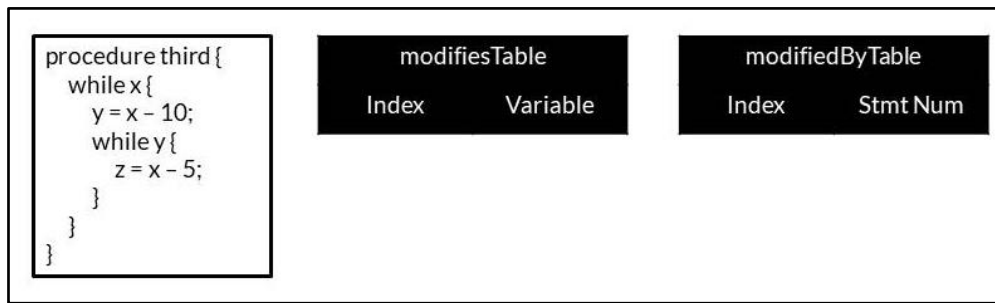
UML diagram of setter functions in Modify class

The function setModifies, called in the PKB, implicitly calls setModifiedBy. This function parses in three arguments: (1) the statement number, (2) the variables being modified, and (3) a vector<int> of the parentStar of the statement number. As the variable is of string type, and the tables in Modify class take in integers, the PKB gets / creates the / an index of the variable, and parses the integer into the function.

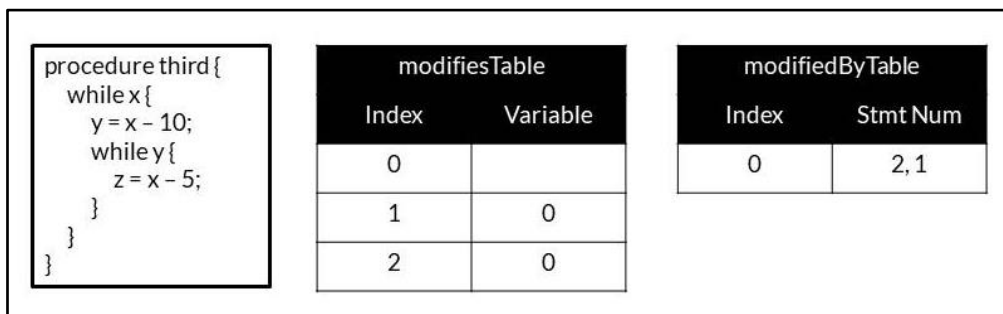
setModifies will store arg2 at index location arg1 in the modifiesTable, and implicitly store arg1 at index location arg2 in the modifiedByTable. Here, Modify class will check parentStar of arg1; if the returned vector is not empty, the values in the vector (i.e. parents of arg1) will also be set to modify arg2.

Example Code 1 (aid visualisation of Modify class table population)

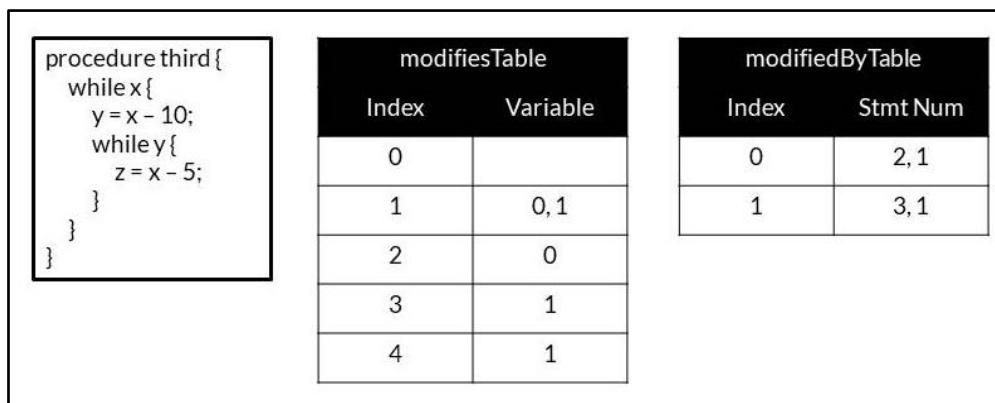
- (1) As line 1 is not an assign statement type, and has no parent, it stores no values.



- (2) Line 2 calls setModifies(2, 0) (i.e. variable “y” was given the index 0 in the PKB) and implicitly does setModifiedBy(0, 2). getParentStar(2) will return 1, thus setModifies(1, 0) is called.



- (3) Line 3 is not of an assign statement type, and thus does not modify variables.
- (4) Line 4 calls setModifies(4, 1) (variable “z” given the index 1 in the PKB) and implicitly does setModifiedBy(1, 4). As getParentStar(4) return 3 and 1, setModifies(3, 1) and setModifies(1, 1) are called.



In the case of Modify, there can also be a relationship between procedure and variable. To set procedure modifies is a separate method (i.e. setProcModifies), and called in the PKB parsing five arguments, (1) the procedure the statement is in, (2) the variable being modified, and (3) procedures called by the current procedure, (4) procedures the current procedure is calling, and (5) procedures the statement is calling.

To better understand the population of the relationship tables when multiple procedures are involved, visual aid will be used.

Example Code 2 (aid visualisation of Modify class table population)

There are two scenario when we need to populate the Modify class for another procedure, (1) when parsing an assignment statement and the current procedure has been called by another procedure, and (2) when parsing a call statement and the procedure being called has already been processed, and Modify class has its table populated already.

- (1) During parsing, when a statement that modifies a variable is encountered (i.e. assign statement), we first check if the current procedure is called by any other procedure by getting calledBy*, and set these procedures to modify the current variable. After which we find out the call statements that call* the current procedure, and along with their parents, modify the variable.
- (2) During parsing, when encounter a call statement is encountered, we first find out the which variables are being modified in the procedure that is being called. Then similar to (1) we find if the current procedure is calledBy* other procedures, and populate the Modify table in the exact way as (1)

Call Class

Call stores relationships between procedures. The class contains four tables - (1) call, (2) calledBy, (3) callStar, and (4) calledByStar. Bi-directional storage of these relationships allows easy response to queries such as Call(c, "second") easily; we call getCalledBy("second").

Call	
<ul style="list-style-type: none"> - callsTable : vector<set<int>> - calledByTable : vector<set<int>> - callsStarTable : vector<set<int>> - calledByStarTable : vector<set<int>> 	
<ul style="list-style-type: none"> + setCalls(int statementNum, int procName1, int procName2) : void - setCalledBy(int procName1, int procName2) : void - setCallsStar(int procName1, int procName2) : void 	

<ul style="list-style-type: none"> - setCalledByStar(int procName1, int procName2) : void
--

UML diagram of setter functions in Call class

The function setCalls (and setCalledBy, done implicitly), called in the PKB, parses in three arguments: (1) the statement number, (2) the procedure calling, and (3) the procedure being called. As the procedures are of string type and the tables in Call class take in integers, the PKB gets / creates the / an index of the procedures , and parses the integer into the function.

setCalls will store arg2 at index location arg1 in the callsTable, and implicitly store arg1 at index location arg2 in the calledByTable. Here, Call class will also setCall* and setCalledByStar implicitly, by finding out what calls* arg1 and setting them to calls* arg2 and vice versa.

Once complete, setCalls will then populate the Modify and Use class by looping through the variables arg2 used / modified, and set arg1 to use / modify, while also populating the call statement and procedures that call arg1.

Recursion check

As calling of procedures must not end up being recursive, an isRecursive check is done at every setCalls. This check is done by looping through the calls table, ensure that at the index location of the table, a procedure of the same index does not exists (i.e. at index location 1 of the calls table, integer 1 should not exists, as this indicates procedure 1 is recursive).

Next Class

Next stores relationships within procedures. The class contains two tables, (1) next, and (2) previous. Bi-directional storage of these relationships allows easy response to queries such as next(a, "10") easily; we call getPrevious("10").

Next	
<ul style="list-style-type: none"> - nextTable : vector<vector<int>> - previousTable : vector<vector<int>> 	
<ul style="list-style-type: none"> + setNext(int stmtNum1, int stmtNum2) : void - setPrevious(int stmtNum1, int stmtNum2) : void 	

UML diagram of setter functions in Next class

The Next class is populated using the PKB method createCFG. This method is called at the end of parsing, and uses the information stored by the parser. By looping through the statements, according to their type (i.e. assign / call / if / while), different recursive functions are called to populate the Next class. In the case of assign / call, the function setNext(current line, next line) is called (implicitly calling setPrevious). In the case of while, the while statement is stored to set the last line of the container back to the while statement; to accomplish this, it is necessary to recursively process the container statements until the last line(s) of the while container is obtained. The same logic applies to the case of if, where this time there are two containers instead of one.

Non-precomputed Relationships

Due to the requirements of the project, these 3 types of relationships could not be precomputed: (1) Next*, (2) Affects and (3) Affects*. All three of them utilize the Next relationship and the CFG, classifying them as graph problem with different constraints. All three relationship share similar parameters where they can have two arguments, which will return a boolean, or one argument, which will return a vector of statements, depending on the position of the argument, and finally no arguments, which will return a pair of vectors that correspond to the two arguments.

Next* Class

For Next*, the algorithm is simpler than the rest as we only need to find out if a statement can be reached through the CFG from a starting statement. Since there are no edge weights in the CFG, we use BFS(fastest) to traverse the CFG and return the answer.

Affects Class

For Affects, the algorithm involves checking if the 2nd statement uses the variable that the 1st statement modifies, without the variable being modified along the path on the CFG. Similarly, we use BFS for the cases with 1 or 2 arguments. We use the Last Modified Table method for the 0 argument case, where we keep track of what was modified at every assign statement and output the relationship whenever we process a new assign statement. Due to the cyclic nature of the CFG, we need to check that the modified tables have changed before exploring the path. This makes the code finite in nature, as eventually, all statements will only need to be processed twice in a while loop once in everything else. (Take note that a statement will need to be processed four times if it is in two while loops, meaning that the number of processing required accumulates as the loops go deeper.

Affects* Class

For Affects*, the algorithm is the most complicated, however, similar to affect, the case with 1 or 2 arguments, we are able to keep a Last Modified Table and use BFS to traverse, as we have a starting point. However, there were some difficulty in ensuring the 0 argument case was 100% correct. It was too late when we realised that we had to combine the Last Modified Table together when we exit the if and else container. Thus, based on the argument above that a statement will only need to be traversed twice in a while loop, we took note of the max depth looping and set $2^{\wedge} \text{max depth}$ as the upper bound of traversing the statement, afterwhich the algorithm will not explore the statement any further. We admit this is not the best way of implementation, but due to time constraints and especially lack of manpower, we have no choice but to do this.

Design Extractor

Design Extractor is required to handle cases where settings done to one class object dominoes to another. However, this is handled by the class that the set function domino-ed to, and not set as a separate class on its own in our SPA.

For Parent and Follow, the design extraction is done when setters implicitly calls star setters, without requiring the parser to call the setters for Parent* and follow*

For Uses and Modify, the design extraction is done when the setters take in the Parent* vector of the statement and sets them to use / modify the same variable implicitly.

Using the Call class as an example, once setCalls has stored the call relationship, it will then begin to populate the Modify and Use class (as mentioned in detail in the Call class section above). Our SPA uses this method to implement design extraction.

For Next, design extraction is done when createCFG method loops through the statement type table instead of reparsing the SIMPLE code. Using the information found through parsing, population of the Next Class tables in one loop of the statement type table is possible.

For Next*, Affect, and Affect*, design extraction is done when these algorithms uses the CFG from Next class and checks the statement types of each statement from PKB.

3.1.3 Query Preprocessor

The preprocessor consists of the components: Relationship, RelationshipTable, synonymEntityPair, synonymAndEntityList, QueryElement, QueryStatement, QueryValidator .

RelationshipTable

- (1) Relationship Object : holds permitted design entities and types for each argument. Multiple Relationship objects are used in the RelationshipTable(used for having quick access to whether a particular argument is permitted in a relationship) and how Relationship objects have three parameters, (1) the number of arguments, (2) the first argument, and (3) the second argument.

Example : Relationship Modifies

Permitted design entities / type arg1: 'stmt' | 'assign' | 'while' | 'prog_line' | 'if' | 'call' | 'procedure' | 'string' | 'number'

Note that 'number' and 'string' is not really a design entity but is a 'type' that is allowed in arg1.

Permitted design entities / type arg2: 'variable' | 'string' | 'wildcard'

```
modifiesArg1 = { "stmt", "assign", "while", "prog_line", "if", "call", "procedure",  
                "string", "number" };  
modifiesArg2 = { "variable", "string", "wildcard" };  
Relationship relationModifies = Relationship(NUM_TWO, modifiesArg1, modifiesArg2);
```

- (2) RelationshipTable Object : object holds all information on arguments allowed for every relationship; RelationshipTable : unorderedMap<string, Relationship>

Using the previous example : Relationship Modifies

```
relationshipTable[MODIFIES_STRING] = relationModifies;
```

The relationshipTable will map (using unorderedmap) these argument types to Modifies for validation. This method / data structure is chosen based on its ease of validation and extendability (ability to easily add / delete any particular type).

SynonymEntityList

- (1) synonymEntityPair Object : holds details on synonyms declared as what design entity it is.

Example query (declaration) : stmt sOne, sTwo;

```
design entity: stmt  
vectors of synonym associated with stmt: { "sOne", "sTwo" }
```

Thus, in a synonymEntityPair object, a design entity: stmt will be associated with a vector of synonyms.

- (2) synonymAndEntityList Object : holds vector of synonymEntityPair objects

Example query declaration portion only : stmt sOne, sTwo; while wOne; if ifsOne;

```

index [0] of synonymAndEntityList Object : (stmt { "sOne" , "sTwo" })
index [1] of synonymAndEntityList Object : (while { "wOne" })
index [2] of synonymAndEntityList Object : (if { "ifsOne"})

```

QueryElement

queryElement Object : stores information on either a clause (such that / pattern / with) or the Select portion of the query. This object is overloaded, and has up to 4 different constructors - one each for (1) Select, (2) with, (3) pattern, and (4) such that.

Example query for queryElement / queryStatement : stmt s; procedure p; assign a; Select s such that Follows*(s,4) with p.procName = "Second" pattern a(,_"x+y"_)

- (1) queryElement contains Select:

```

entity: stmt      synonym: s      Type: synonym      synAttr: synonym

```

- (a) Example involving tuple : procedure p; stmt s; Select <p.procName, s>

```

procedure p; stmt s; Select <p.procName, s>
entity: procedure, stmt      synonym: p,s
Type: tuple                  synAttr: procName, synonym

```

- (2) queryElement contains Follows*(s,4)

```

Follows*(s,4)
argument1 = s      arg1Type: synonym      arg1Entity: stmt
argument2 = 4      arg2Type: number        arg2Entity: empty
relationship = Follows*      clause = suchThat

```

- (3) queryElement contains p.procName = "Second"

```

with p.procName = "Second"
arg1 = p.procName      arg1Type = procName      arg1Entity = procedure      arg1Synonym = p
arg2 = Second          arg2Type = stringLiteral  arg2Entity = ""             arg2Synonym = ""
clause = with

```

- (4) queryElement contains pattern a(,_"x+y"_)

```

pattern a(, _"x+y"_)
patternEntity = assign  patternSynonym = a
arg1 = _                arg1Type = wildcard    arg1Entity = empty
arg2 = x+y              arg2Type = substring
arg3 = empty            arg3Type = empty
clause = pattern

```

QueryStatement

QueryStatement Object : serves as the main object for storing, declarations, QueryElement Objects and synonymEntityList objects. This object contains organised information that will be passed to the evaluator for evaluation.

Visual representation of queryStatement's private attributes:

```

QueryElement selectElement;
vector<QueryElement> withElement;
vector<SynonymEntityPair> synonymEntityList;
vector<QueryElement> normalQueryElements;
vector<QueryElement> hardQueryElements;
multimap<string, pair<int, int>> normalMultiMap;
multimap<string, pair<int, int>> hardMultiMap;
bool invalidQuery;

```

Brief outline of private attributes of the QueryStatement:

- selectElement contains all required information on the result-clause of the Select query as shown in (1) under the queryElement section
- withElement contains all required information on all the “with” clauses in the query, and contains a vector of queryElement as shown in (3) under the queryElement section.
- synonymEntityList contains all required information on all the synonyms that are declared along with its associated design entities
- normalQueryElements contains all required information on all “such that” and pattern clauses, excluding hard clauses (defined as: Next* / Affects / Affects* clauses). This vector can contain queryElement of types (2) and (4).
- hardQueryElements contains all required information on all hard clauses; it is stored as a vector. This vector can contain queryElement of type (2) only.
- normalMultiMap is a map that provides quick access to information (argument number and index number) on the synonyms used in normalQueryElements
- hardMultiMap is a map that provides quick access to information (argument number and index number) on the synonyms used in hardQueryElements..

Further illustration on how normalQueryElements, normalMultiMap, hardQueryElements and hardMultiMap is stored for the following query:

stmt s1, s2; assign a; Select s1 such that Next*(s1, s2) and Parent(s2, a)

```

normalMultiMap - [0] : Parent(s2,a)
key: ["a"] value: ["2", "0"]
where the key 'a' is the declared synonym a and allows quick access to value: argumentNumber '2'
and queryElement index number '0' (index number is clause/queryElement index in normalQueryElement)
key: ["s2"] value: ["1", "0"]
where the key 's2' is the declared synonym s2 and allows quick access to value: argumentNumber '1'
and queryElement index number '0' (index number is clause/queryElement index in normalQueryElement)

hardMultiMap - [0] : Next*(s1,s2)
key: ["s1"] value: ["1", "0"]
where the key 's1' is the declared synonym s1 and allows quick access to value: argumentNumber '1'
and queryElement index number '0' (index number is clause/queryElement index in hardQueryElement)
key: ["s2"] value: ["2", "0"]
where the key 's1' is the declared synonym s2 and allows quick access to value: argumentNumber '1'
and queryElement index number '0' (index number is clause/queryElement index in hardQueryElement)

```

QueryValidator

QueryValidator Object : parses and validates the incoming query line; holds private attributes required for validation, which will be passed to the evaluator, namely: synonymAndEntityList, QueryStatement, RelationshipTable

Procedure for parsing and validating queries:

Step	Process
1	<ul style="list-style-type: none"> a tokenizer is used to parse queries (using semi-colons and synonyms, with its design entities); the declared design entity is stored as a string query object. the string query object (entity to be checked) is parsed through validEntities (a vector) for validation of declarations. ValidEntities contains all valid design entities. synonymEntityPair object is instantiated and added to a synonymAndEntityList object.
2	<ul style="list-style-type: none"> check if query is Select synonym or BOOLEAN or attrRef or tuple look for keyword (pattern / such that) in the first clause (i.e. immediately after the Select statement) Validation is done by checking if the string(synonym/BOOLEAN/attrRef/tuple) before the keyword exists in

	<p>synonymAndEntityList,</p> <ul style="list-style-type: none"> ○ If synonym, simply check if keyword/synonym exists in synonymAndEntityList. ○ if BOOLEAN, check for an exact string match. ○ if attrRef, extract the design entity, and perform validation as mentioned in synonym validation. ○ if tuple, check if all argument delimited by commas are valid synonyms or attrRef <ul style="list-style-type: none"> ● once validation is complete, a queryElement is created for the Select portion of the query, and added to the queryStatement under selectElement attribute.
3	<ul style="list-style-type: none"> ● current string is split further into substrings by keywords (e.g. such that), and placed into a vector of strings ● the vector will be iterated through to check for the type of clause (such that / pattern / with) at each index
3a	<p>For such-that clauses,</p> <ul style="list-style-type: none"> ● first validation is done using isValidSuchThat (contains methods isValidSuchThatRegex and isValidSuchThatExtendedRegex) <p>Example:</p> <ul style="list-style-type: none"> - such that Follows(s,4) and Modifies("First", a) is valid - such that Follows(s,4)Modifies("First",a) is invalid. <ul style="list-style-type: none"> ● extract clauses with extractSuchThat and place them in a vector <p>Example:</p> <ul style="list-style-type: none"> - such that Follows(s,4) and Modifies("First", a) will be stored as [0]: Follows(s,4) [1]: Modifies("First", a) <ul style="list-style-type: none"> ● a second validation is done using relationshipTable to check query relationship validity

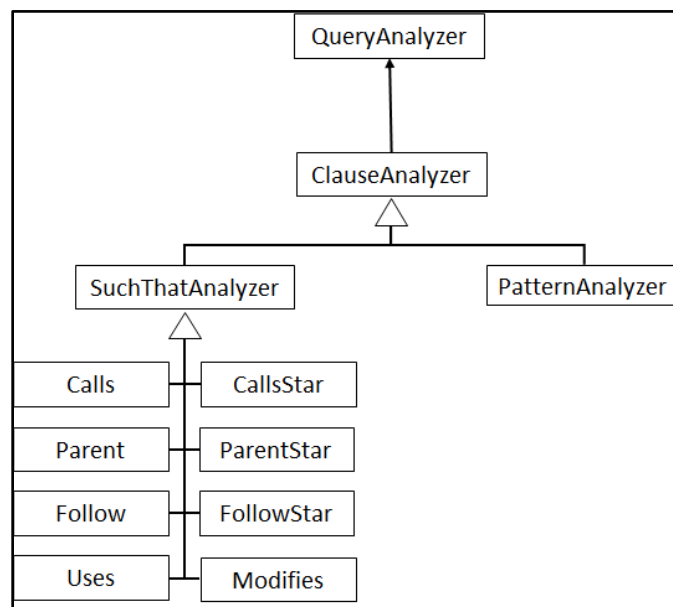
	<p>Example:</p> <ul style="list-style-type: none"> - such that Follows(s,4) is valid as arguments 1 and 2 are permitted arguments of Follows relationship - such that Follows("xor", "nor") is invalid as arguments 1 and 2 are not permitted arguments of Follows relationship • at this stage, query validation is complete. A queryElement is created and added to queryStatement under private attribute normalQueryElement or hardQueryElement (only Next* / Affects / Affects*)
3b	<p>For pattern clauses,</p> <ul style="list-style-type: none"> • extractPattern is used and validated using isValidPattern • validation asserts expected SIMPLE grammar standards <p>Example:</p> <ul style="list-style-type: none"> - pattern a(,"x") is valid - pattern a2 a(,"x") is invalid • check design entity before '(' (it can only be 'assign' 'while' 'if'); validation is done using isValidAddAssignPattern, isValidAddWhilePattern, isValidAddIfPattern accordingly <p>(1) isValidAddAssignPattern:</p> <ul style="list-style-type: none"> - Permitted design entities arg1: 'synonym' '_' 'string' 'number' - Permitted design entities arg2: '_' '_expr_' 'expr' - expr is validated using isExpression (in Util class) <p>(2) isValidWhilePattern:</p> <ul style="list-style-type: none"> - Permitted design entities arg1: 'synonym' '_' 'string' 'number' - Permitted design entities arg2: '_' <p>(3) isValidIfPattern</p> <ul style="list-style-type: none"> - Permitted design entities arg1: 'synonym' '_' 'string' 'number' - Permitted design entities arg2 / arg3: '_' <ul style="list-style-type: none"> • once validation is complete, a queryElement is created and added to

	queryStatement under normalQueryElement attribute.
3c	<p>For with clauses,</p> <ul style="list-style-type: none"> validation is done using isValidWith (contains methods isValidWithRegex and isValidWithExtendedRegex) <p>Example:</p> <ul style="list-style-type: none"> with p.procName = "First" and n= 10 is valid with p.prOcNam3 = "First" and n= 10 is invalid extract clauses with extractWith and place them in a vector <p>Example: with p.procName = "First" and n = 10</p> <p>[0]: p.procName = "First"</p> <p>[1]: n = 10</p> <ul style="list-style-type: none"> split the string to obtain arguments 1 and 2, and validate them (both arguments must be the same, either of type 'string' or 'number') using withClauseBankType. During this validation process, attrNameBank is also used. <p>withClauseBankType:</p> <p>A map which matches the type of argument to its specific value type.</p> <pre>withClauseTypeBank[PROCNAME] = STRTYPE; withClauseTypeBank[VARNAME] = STRTYPE; withClauseTypeBank[VALUE] = INTTYPE; withClauseTypeBank[STMTNUM] = INTTYPE; withClauseTypeBank[PROG_LINE_STRING] = INTTYPE; withClauseTypeBank[STRING_LITERAL] = STRTYPE; withClauseTypeBank[NUMBER_STRING] = INTTYPE;</pre> <p>Example:</p> <p>withClauseTypeBank[procName] gets the value of "strType"</p> <p>withClauseTypeBank[value] gets the value of "intType"</p> <p>Example:</p> <p>p.procName = "First" is valid as both sides are of "string" type</p> <p>p.procName = c.value is invalid as both sides are not of the same type</p>

	<p>attrNameBank:</p> <p>A map used for cases where a “with” clause argument involves an attrRef (defined in the SPA grammar handbook); also useful for validation.</p> <pre>attrNameBank[PROCNAME] = { PROCEDURE_STRING, CALL_STRING }; attrNameBank[VARNAME] = { VARIABLE_STRING }; attrNameBank[VALUE] = { CONSTANT_STRING }; attrNameBank[STMTNUM] = { STMT_STRING, ASSIGN_STRING, IF_STRING, WHILE_STRING, CALL_STRING };</pre> <p>Example:</p> <p>p in p.procName is a reference to a procedure; attrNameBank maps p using procName as key, and checks if the procedure exists.</p> <ul style="list-style-type: none"> once validation is complete, a queryElement is created and added to queryStatement under withQueryElement attribute.
4	<p>getQueryStatement() is called by evaluator to begin evaluation which returns the whole query Object</p>

3.1.4 Query Evaluator

Query Evaluator Architecture



The evaluator uses Abstract Factory Pattern to handle the various clauses (“with” / Affects, etc.), reducing code redundancy and allowing easy addition of new clauses (scalability) – adhering to the open–close principle.

The evaluator consists of the components stated below,

clauseAnalyzer : factory that generates the relevant clause solver object given a query. It takes in a single clause object as the parameter, and returns clauseResult

clauseResult : a 2-dimensional vector<string> containing results of a single query

WithAnalyzer, PatternAnalyzer, SuchThatAnalyzer : classes contained, and are used, by clauseAnalyzer. These components take in valid queries as arguments, and requests information from the PKB to produce an AnalyzerObject, returning a boolean and a list of vectors containing the queries' results.

suchThatAnalyzer : parent component of normal and hard clauses. Used to categorize clauses based on whether the query results can be pre-computed. Returns clauseResult containing up to two vectors: (1) one for each synonym in argument 1 and 2 and (2) a boolean hasSuchThat to clauseAnalyzer

- Normal clauses are defined as : Follows, Follows*, Parent, Parent*, Uses, UsesProc, Modifies, ModifiesProc, Calls, Calls*.
- Hard clauses are defined as : Next*, Affects, Affects*

patternAnalyzer : handles pattern queries (assign / if-then-else / while). Returns clauseResult containing two vectors, (1) one for each synonym in argument 1 and 2 and (2) a boolean hasPattern clause to clauseAnalyzer.

Example:

- pattern arg1(arg2,exp) for assignments
- pattern arg1(arg2,_,_) for if-else, Pattern arg1(arg2,_) for while

withAnalyzer : handles query objects and checks for attribute equivalence, such as synonym.attrName = synonym.attrName, and synonym = synonym.attrName. Returns clauseResult containing two vectors. (1) one for each synonym in argument 1 and 2 and (2) a boolean hasWithClause to clauseAnalyzer.

Example:

- arg1.attrName = arg2.attrName

AnalyzerObject : an object created to solve either normal / hard clauses

queryAnalyzer

queryAnalyzer : retrieves the clause results from clauseAnalyzer and updates instance variables of queryAnalyzer

hasClauseType : “with” / pattern/ “suchThat” : ensures boolean result of all clause types are true before evaluating the next query object. If any boolean indicator is false, queryAnalyzer returns a boolean value if Select is boolean, or an empty vector otherwise.

mergedQuerytable (instance variable) : intermediate table which maintains the latest result of analyzed query objects. If clauseResult obtained through clauseAnalyzer has no synonyms in common with the intermediate table, the results are added into a separate table index.

Otherwise, a hash-join operation, explained later, is performed to combine the results from clauseAnalyzer. This way, cartesian product for synonyms are minimized.

- mergedQueryTable is a 3-dimensional vector<string>, representationing a list of synonym tables

clauseSynonymMap (instance variable) : a map using the synonym as key and the value as a tuple, (index of the clauseResult, location of synonym in clauseResult)

selectSynonymMap (instance variable) : a map for the Select synonym. Used to determine if synonyms in a tuple are from the same table index in the mergedQueryTable. If synonyms are from the same table, the results are concatenated into a string before cartesian product operation is applied. Using this map, cartesian product results can be re-ordered to follow the tuple ordering.

Example:

- synonym A, B are from Table 1 in mergedQuery Table, C is from Table 2, and D is from Table 3
- Suppose Select <A,D,B,C> , the selectTuple function in queryAnalyzer will perform a cartesian product with AB, D, and C . The ordering here is incorrent, and selectSynonymMap is used to determine the indices of the results that require swapping.

selectSynonymTable (instance variable) : a 2-dimensional table, where rows represent synonyms belonging in the same table index inside mergedQuerytable, and columns represents the concatenated synonym results for that particular table index

queryOptimizer

clauseRanker : datastructure that containing 5 attributes:

- (1) single query clause object
- (2) rank of synonyms
- (3) rank of relationship clauses (i.e. "with" / "suchThat" / pattern)
- (4) rank of design entities (i.e. assign / if / with)
- (5) rank based on the number of synonyms in the clause object

A custom comparator is used to sort the list of query objects. Priorities are set as follows,

- (1) sort by number of synonyms : rank 1 = 0 synonyms, rank 2 = 1 synonym, and rank 3 = 2 synonyms
- (2) if there is a need for a tie-breaker, sort by rank of relationship clauses - rank determined by their respective table size (hard clauses (next*, affects, affects*) are assigned ranks 9, 10, 11 respectively)
- (3) if there is a need for a tie-breaker, sort by rank of design entities - rank is based on the maximum possible results for each designEntity.

Example:

- for stmt# and prog_line : statement numbers from 1 to n, where n is the last statement in the simple program
- (4) if there is still a need for a tie-breaker, sort by the frequency of the synonyms appearing in the queryStatement object
 - (5) if there is still a tie, return the first clause being compared

Intermediate Table Result - MergedQueryTable

The intermediate table is represented by a list of tables which we implemented as a 3D array which is `vector<vector<vector<string>>>` in C++. Each vector represents all the result for a single synonym. The vector class was chosen due to the ease of maintaining the results from evaluating clauses and intersection between different tables.

We chose to insert the synonym at the end of each vector to maintain the synonym map and for merging purposes. By using a synonym map - `unordered_map<string, tuple<int,int>>`, we can determine the vector location in $O(1)$ by hashing the synonym with the table location, vector location as value. Since we inserted the element at the end of the vector, we can remove it in $O(1)$ time during the evaluation of the select clause.

However, the downside is the vectors cannot be sorted easily. For example the result $\text{Parent}(A, B)$, the values are stored in a column vector and not a row vector. Hence, the values are inserted into the vector A and vector B in the same sequence. Therefore if the vectors are to be sorted, a custom function needs to be implemented otherwise the relation between the two vectors may be incorrect. Since, the order of values is not crucial and our sorting algorithm did not require sorting, we accepted this drawback.

The 2D vector is used to indicate that all synonym results in that table connected directly/indirectly by a relationship clause. When a new clause is evaluated, we check if it shares any common synonym in the current table, if it does we will merge it otherwise we store it in a different location inside the intermediate table.

Current	
Index:0	
1	4
2	5
3	6
A	B

Incoming	
Index:1	
7	10
8	11
9	12
14	13
C	D

Map
A, <0,0>
B, <0,1>
C, <1,0>
D, <1,1>

Hash-join (merging) algorithm

Instead of doing a cross product when the new result of a clause is added to the intermediate query table, we only do it when there exist at least one common synonym between the clause result and the intermediate table.

An example is shown below when an incoming clause contains one synonym.


Incoming		Current			Map
		Index:0			A, <0,0>
4	24	1	4	24	B, <0,1>
6	25	3	6	25	F<0,2>
B	F	A	B	F	C, <1,0>
					D, <1,1>

Compared to the other merge join algorithms , the nested loop is $O(NM)$, merge-sort is $O((N+M) \log (N+M))$ while hash-join is $O(N+M)$. where N = number of elements in 1st vector and M = number of elements in the 2nd vector.

For the hash-join, we first hash the values of the join column in the 2nd table - the common synonym between the two tables. Then we lookup these values in the 1st table and if the value is found, we push these values into a temporary 2D table. Once all elements in the 1st table have been read, we replace the 1st table with the temporary table.

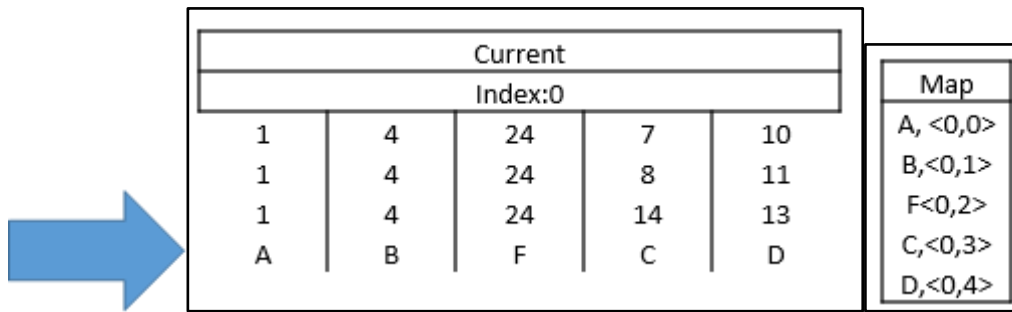
Suppose, there incoming clause result has 2 synonyms residing in different locations of the intermediate table, we will first hash join on the 1st synonym. We then take the result of the first hash join and merge it with the 2nd table inside the intermediate table. After that, we will erase the 2nd table from the intermediate table.

Incoming	
7	24
8	24
14	24
C	F



Incoming		
Index:1		
7	10	24
8	11	24
14	13	24
C	D	F

Map
A, <0,0>
B, <0,1>
C, <1,0>
D, <1,1>
F, <1,2>



If the index of the 2nd table in the 3D table is less than the 1st table, we will replace the 2nd table with the merged result and delete the 1st table. Otherwise, we will replace the 1st table with the merged result and delete the 2nd table.

Then we will re-map all synonym in the Map with table locations greater than the deleted index. Thus this will take up to $O(V)$, where v = number of synonyms in the hash table.

Query Evaluator Process

On initialization, evaluator takes in a queryStatement, containing vectors of queryElement objects, creates a PKB pointer to the PKB class, and all required maps (i.e. selectSynonymMap, clauseSynonymMap, etc.).

Procedure for evaluation:

Step	Process
1	<ul style="list-style-type: none"> evaluator uses the getNormalQueryElements() and getHardQueryElements() api to obtain clauses required for evaluation information is then passed on to queryOptimizer, which assists with approximating the optimal order for analyzing the various clauses based on the heuristics indicated in custom comparator
2	once sorted clause order is received, queryEvaluator proceeds to ask clauseAnalyzer for results of each clause
2a	<ul style="list-style-type: none"> clauseAnalyzer checks the clause type to be produced (e.g. pattern / “such that” / with) once clause type and sub-relation (if / while / assign for pattern, or

	<p>modifies, etc. for “such that”) has been determined, analyzerObject will execute the solve method to obtain clauseResult</p> <ul style="list-style-type: none"> as PKB gives statement number results for all statement types for each relationship, the evaluator needs to ensure values are restricted before caching the results. Hence, clause results are validated against their design entities to restrict the number of values before caching them. <p>Example: stmt s1,s2 ; while w1;</p> <ul style="list-style-type: none"> Follows(s1,w1). PKB will return results equivalent to Follows(s1,s2)
2b	<p>For the case of Pattern,</p> <p>Example:</p> <ul style="list-style-type: none"> pattern arg1(arg2 , “patexp”) for assign pattern arg1(arg2, _ , _) for if-else pattern arg1(arg2, _) for while <ul style="list-style-type: none"> analyzerObject checks the contents of arg1 ,arg2, arg1Entity, arg2Entity and patternType <p>For assign patterns,</p> <ul style="list-style-type: none"> pass to PKB “patexp” and obtain pattern result getPatternVariable for wildcards getPatternExpression for exact expression getPatternExpressionSubString for sub-string expression getPatternVariableExpressionSubstring for when arg2 = literal and sub-string expression getPatternVariableExpression for when arg2 = literal and “exactexpression” <ul style="list-style-type: none"> for if-then-else and while patterns, pass to PKB arg2 as an argument and obtain pattern results getPatternIf getPatternWhile <ul style="list-style-type: none"> if clauseResult is non-empty, boolean TRUE will be returned, and

	<p>clauseResult is passed to ClauseAnalyzer. Otherwise, analyzerObject returns FALSE with an empty vector.</p> <p>For the case of “with”,</p> <ul style="list-style-type: none"> • Check existence of equality between two given refs. • if equality between two refs holds, return TRUE with the clauseResult, and FALSE with empty vector otherwise <p>Example: Call c, procedure p</p> <ul style="list-style-type: none"> - c.procName = p.procName • check existence of Call statement in the program, with call statement calling procedure p. • clauseResult will contain 2 vectors, (1) Call statement number and (2) procedure being called <p>For the case of “such that”,</p> <p>Case: Normal Clause</p> <ul style="list-style-type: none"> • clauses that allow results to be precomputed and cached • given rel(arg1,arg2),aAnalyzerObject checks under which of these 9 cases a query falls under <ul style="list-style-type: none"> - case 1 : literal, literal - case 2 : literal, synonym - case 3 : literal, wildcard - case 4 : synonym, literal - case 5 : synonym, synonym - case 6 : synonym, wildcard - case 7 : wildcard, literal - case 8 : wildcard, synonym - case 9 : wildcard, wildcard • assuming synonyms arg1 and arg2 have not been evaluated in a previous query, analyzerObject will evaluate the clause by querying the PKB using a vector containing all possible values; otherwise it will use previously
--	--

	<p>cached results and relevant API.</p> <p>Example: Call c1,c2; variable v1;</p> <ul style="list-style-type: none"> - Calls(c1,c2) Modifies(c1,v1) <ul style="list-style-type: none"> ○ suppose Call query is evaluated before the Modifies query, calls(c1,c2) will use the api getCalls(arg1) for all possible arg1 ○ It then stores the result into two vectors inside clauseResult ○ once information from analyzerObject has propagated to QueryAnalyzer, it will be stored in the intermediate table called mergedQueryTable <ul style="list-style-type: none"> ● however,in the case of Modifies, as c1 has been evaluated before and can be found inside mergedQuerytable, getModifies(c1) can be used to determine the pair of all modifies (c1,v1), instead of using all call statements. ● In this case, time taken to analyze each query when the synonym has been evaluated before is amortized <p>Case: hard clause</p> <ul style="list-style-type: none"> ● clauses not permitted to be pre-computed and cached ● In this case, results are not stored in any PKB table. The fastest way to compute the result is to call the relevant API even if synonym has been previously evaluated. <p>Example: Call c1,c2; stmt s1;</p> <ul style="list-style-type: none"> - Calls(c1,c2) Next*(c1, s1) <ul style="list-style-type: none"> ● Though c1 has been evaluated in the Call query, time taken to execute the API getNextStarTwoSynonyms() is faster than executing getNextStar(arg1) for all arg1 unique results in the mergedQueryTable by almost ten times
3	<p>For the case of caching / merging clauseResults,</p> <ul style="list-style-type: none"> ● result is cached by first performing a hash-join operation. ● If intersection of results is empty, we stop evaluating any further and return a boolean result if it is Select boolean, and an empty vector otherwise.

4	<p>For selecting the results,</p> <p>Case: single synonym</p> <ul style="list-style-type: none"> • if all boolean instance variables (i.e. hasWith, hasPattern, hasSuchThat) are TRUE, a search will be conducted on SelectSynonymMap for the queried synonym. • if any boolean instance variables is FALSE, an empty result is returned from queryAnalyzer to the UI / Autotester • if synonym can be found, result will be retrieved from mergedQueryTable. Otherwise, results for the synonym's design entity is returned instead • a corner case to highlight will be for calls.attributeName. As calls can either be selected for a statement number or procedure name, queryAnalyzer will do an additional check to see if there is a requested attrName for the select synonym. If there is such a request, the statement number will be converted to its procedure using PKB's API getProcCalledByStatement. <p>Case: boolean</p> <ul style="list-style-type: none"> • return true if all boolean instance variables or queryAnalyzer are true, and false otherwise <p>Case: tuple</p> <ul style="list-style-type: none"> • first group synonyms in the tuples based on their table index location inside mergedQueryTable. • next, concatenate the results for each group into a string. Duplicate strings for each group are removed before moving onto the cartesian product. • for the cartesian product, two groups are taken at a time and their cartesian products are computed, until all groups have been evaluated. • in the case where there is only one group, no cartesian product is performed • once cartesian product is complete, synonyms are reordered based on the Select clause ordering (order may have been lost during grouping of synonyms from the same table index inside Synonym Table)
---	---

“Such That” Clauses

Case(Arg1,Arg2)	Method to Solve	Example
Rel (var, var) Validation	Call the appropriate PKB Api and check if the relation exists. Returns a boolean result which is updated in the QueryAnalyzer instance variable.	Parent(1, 5) - returns true if relation is present and update hasSTClause
Rel (var, syn) Evaluation	Call getRel(var). Returns a list of strings satisfying the relation(var, syn) . Insert the results into a column vector with syn.name as the last element for mapping.	Uses(proc1, v) - call ProcUses(proc1) and get all variables which proc1 uses. Uses(1,v) - call Uses(proc1) and get all variables which stmt#=1 uses. proc1 is the last element in the list if the PKB results is >0.
Rel (var, _) Validation	Validates the relationship by checking the size of getRel(var). result is true if the list of results > 0. False otherwise	Modifies(1,_) - check with PKB if stmt 1 modifies any variable
Rel (syn, var) Evaluation	Similar to case (var,syn) but calls getRelBy(var)	Calls*(p1,"hello") - return procedures called by “hello”, and true if list is > 0. hello is the last element in the list
Rel (syn, syn)	Get all values – Result1 associated with the design entity of 1st syn.Then proceed by calling the rel(var, syn) – Result2. Add the value of result1 and result2 into two different lists in the same order. The 1st syn will be the last element of the list for result1 and 2nd syn will be the last element of the list for result 2.	Parent(a1, a2) - get list of all stmts which matches the design entity of a1

Rel (syn, _)	Get all possible values for 2nd argument and call getRelBy(possibleValues) stores the result and removes duplicates	Affects(a1, _) - call AffectsAPI getAffectsTwoSynonyms() , validate design entity of a1 and stores the result only for a1.
Rel (_ , var) validation	Similar to Rel(var, _)	follows*(_, 3) - check if 3 is followedBy some statement. - returns true if relation exists
Rel (_ , syn)	Similar to rel (syn, _)	follows(_, k) - return all stmts followedBy k - returns list with k being the last element if the clause holds true
Rel (_ , _) validation	calls getAllRelation and returns true if size > 0.	Parent*(_, _) - check if there exist at least one element in the result from the pkb call.

Pattern Clauses

Case	Method to solve	Example
Containers	<p>pass in variables to method getPatternWhile(variable) if the design entity of the clause is “while” or getPatternIf(variable) for “if”</p> <p>Retrieve a list of statement numbers where the relationship holds.</p> <p>Returns a tuple consisting of the boolean value, a List of entity synonym and control variable synonym (if applicable).</p>	<p>ifs(_, _, _)</p> <p>- returns all the if statement numbers and ifs is the last element in the entity list, boolean = true</p> <p>ifs(v, _, _)</p> <p>- boolean = true, - entityList = all if statement numbers</p> <p>- variableList = all control variables in the respective if</p>

		statements - v is the last element in the variable list for mapping purpose
Assignment	<p>Call getPattern(variable) and verify the expression obtain from PKB are the same as the query object.</p> <p>There are 3 categories, exact, substring and wildcard denoting the type of comparison to execute.</p> <p>Returns a tuple boolean, entityList, variableList</p>	a(v, _"x+y" _) - calls getPattern(v) and check if the pkb expression contains a substring of the query object expression - adds the statement number if it is true - entityList: adds all statement numbers with 'a' being the last element in the entityList. All these statement numbers have the design entity "assign". - variableList: adds all corresponding variables. return <bool,entityList,variableList>

3.2 Design Decision

3.2.1 Parser

We conduct validation, parsing and populating per statement, without the overhead time complexity of multiple parsing. These concerns, implementation and component testing is handled by a single person, and thus we decided on using a single, instead of multiple (once each for validation, parsing and table population), parse for our program. Using one parse, we are able to terminate quickly when there is an error early in the SIMPLE code, thus short circuiting.

Storing the Parent

Problem	Obtaining correct parent in nested containers
Alternatives	1st Approach: Call <code>pkb.getParent(Parent statement)</code> to get grandparent 2nd Approach: Use a stack, and pop to find the immediate parent
Criteria considered when comparing alternatives	<ul style="list-style-type: none">- Time vs. Space complexity- Scalability & Extendability of code- Readability of code

	1st Approach
Advantages	Lower space complexity
Disadvantages	<ul style="list-style-type: none">- Lower readability of code due to <code>getParent</code> call loops- Difficult to expand project (e.g. a statement can have two if-then-else parents, one being if and the other else)- Unintuitive

Approach chosen: 2nd Approach

Here, time complexity is not considered as both approaches require looping through all parents. Though space is a main concern, the pros of scalability, and the added ability to validate SIMPLE code, outweighed the cons (only in cases with heavy nesting will the stack be large).

Storing the Nesting Level

Problem	Storing nesting information to identify container statements
Alternatives	1st Approach:

	<p>Use boolean to check if in container statement</p> <p>2nd Approach: Use integer to identify current nesting level</p> <p>3rd Approach: Use stack to store the container statements to tell us if we are in a container statement</p>
Criteria considered when comparing alternatives	<ul style="list-style-type: none"> - Time vs. Space complexity - Scalability and Extendability of code - Readability of code

	1st Approach	2nd Approach	3rd Approach
Advantages	<ul style="list-style-type: none"> - Lower space complexity - Lower time complexity 	<ul style="list-style-type: none"> - Intuitive approach - Better readability 	Already implemented with Parent stack
Disadvantages	Difficult to identify when to change value, extra checking required	<ul style="list-style-type: none"> - Higher space complexity (integer vs. boolean) - Integer comparison required 	<ul style="list-style-type: none"> - Not as intuitive - Higher time complexity - requires call to <code>stack.size()</code> before comparison

Approach chosen: 3rd Approach

It is already used in other design decisions, so it's already implemented. There is no competition here, there is 0 space cost. And the slight time complexity is worth it.

3.2.2 PKB

Creating variable and procedure index tables

Problem	Using index vs. string when storing relationships of source code
Alternatives	1st Approach:

	Handle entities as strings 2nd Approach: Designate indexes to them
Criteria considered when comparing alternatives	<ul style="list-style-type: none"> - Search time complexity - Space complexity

Approach chosen: 2nd Approach

Strings have high search time complexity, in comparison to integer (jump straight to index location). An example would be in the case where PKB stores the relationship procedure "Second" modifies variable "x", a map would be required to store and retrieve this information. However, with integers, a vector<int> is sufficient to store this relationship, and retrieval is simpler with the ability to go straight to the queried location in the vector.

Storing of relationships in PKB

Problem	Storing of relationship information for easy retrieval
Alternatives	1st Approach: Create AST 2nd Approach: Create classes of relationship types to contain relationship tables accordingly
Criteria considered when comparing alternatives	<ul style="list-style-type: none"> - Time complexity - Space complexity

Approach chosen: 2nd Approach

Creating an AST would required traversal of the tree during evaluation; this increases evaluation time. Creating relationship tables and handing pre-computed values decreases evaluation time significantly. Though an AST would require less space compared to multiple relationship tables, as time for evaluation is of a higher priority, this approach was chosen.

3.2.3 Query Pre-processor

Changing QueryTree to QueryStatement object

Query tree would require insertion, searching and possibly modifying (for optimization) of nodes, increasing evaluation time. An object-orientated approach to storing query clauses allows for easy modification when a given clause to be enhanced or expanded. For example, the first argument of Follows may be now able to accept a 'procedure' design entity and can be easily implemented.

RelationshipTable & withClauseTypeBank

Problem	Query validation
Alternatives	1st Approach: Hardcode all variations of valid arguments using multiple if-else statements 2nd Approach: Use relationshipTable and withClauseTypeBank (and attrNameBank) to store valid types of relationships of clauses
Criteria considered when comparing alternatives	<ul style="list-style-type: none">- Time vs Space complexity- Scalability and Expandability of code- Readability of code

	1st Approach	2nd Approach
Advantages	<ul style="list-style-type: none">- Easy implementation- Lower space-time complexity	<ul style="list-style-type: none">- Scalable- Higher code readability- Follows separation of concern
Disadvantages	<ul style="list-style-type: none">- Lower code readability (with multiple if-else statements)	Higher space complexity (data pre-computed)

	<ul style="list-style-type: none"> - Difficult to new relationship / clauses (not scalable) - No separation of concern 	
--	--	--

Approach chosen: A combination of the first and second approach

relationshipTable stores permitted arguments of relationships, while withClauseTypeBank stores permitted types for arguments (e.g. procName is of 'string' type) and attrNameBank stores permitted design entity for an attrRef(defined in handbook) type. Pattern requires vigorous checking procedures as it involves the checking of well-formed expression which is difficult to be stored as a 'type' of permitted argument in a table, thus, a table-driven technique was not used here. Scalability of pattern is better with approach one, as it would only require the addition of new methods. Though slight readability is sacrificed, a balance between that, scalability and space-time complexity is achieved with this approach.

3.2.4 Evaluator

Cartesian Product and tuples

By grouping synonyms into the same table inside the intermediate table, we reduce the number of times we perform a cartesian product. However, the price to pay will be reordering of the tuple synonyms during select tuple. Since performing a cartesian product is expensive, we chose to only execute the cartesian product during the select clause if required.

Therefore, for every single cartesian tuple, we have to reorder them based on the original configuration given by the query which will take $O(N)$ time per entry. This means that if there are k entries for the tuple, then it will take a total of $O(NK)$ time which is cheaper than maintaining the full set of synonym cartesian products in the intermediate table.

3.3 Component Interactions

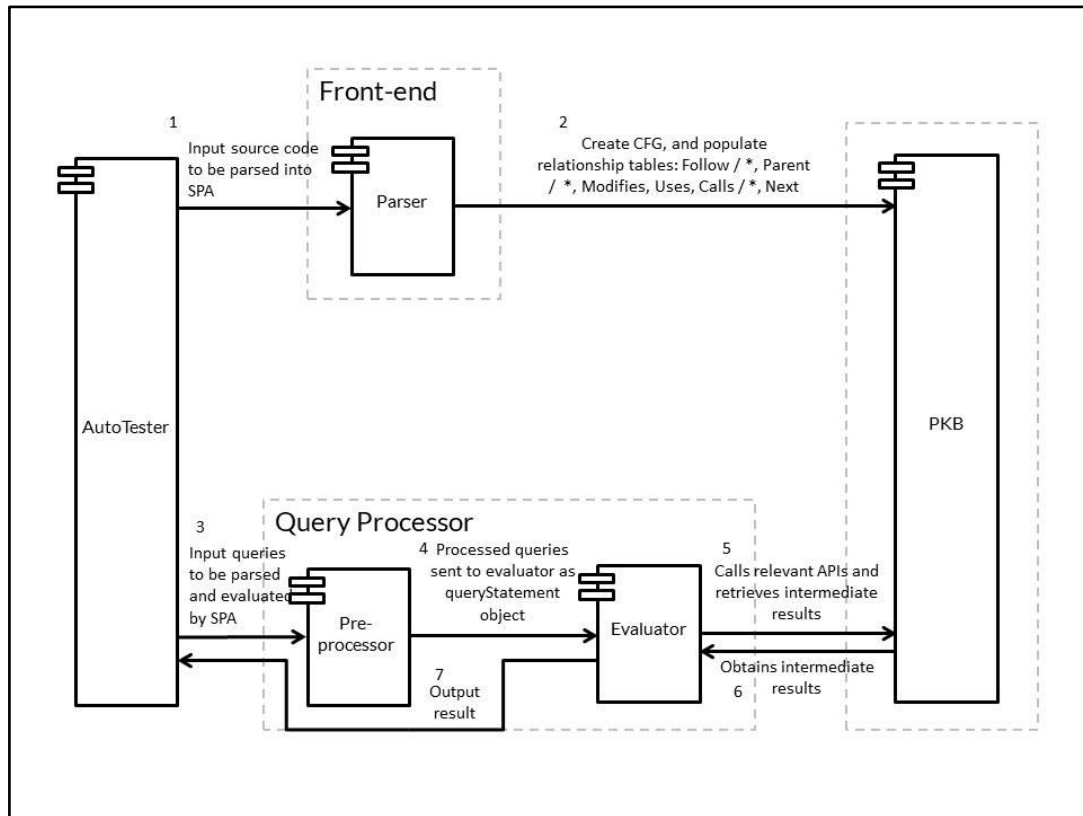


Figure 3.3.1 SPA component interactions

Walkthrough of Component Interactions

- (1) AutoTester inputs a SIMPLE source code to the parser
- (2) The parser, reading the input code line-by-line, validates and calls the corresponding PKB APIs to populate the relationship tables in the program knowledge base (PKB). With the parser's input, the PKB is able to create a CFG.
- (3) The AutoTester inputs queries based on the input SIMPLE source code through the preprocessor.
- (4) The preprocessor validates and parses the queries into queryStatement objects to be read by the evaluator.
- (5) The evaluator receives the queryStatement object, and calls the relevant PKB APIs to retrieve intermediate results of the query.
- (6) The PKB is accessed by the evaluator, and sends requested data.

- (7) The evaluator merges the intermediate tables and forms a final result, which it outputs back to the AutoTester.

4 Documentation and Coding Standards

Github was used for version control while coding the system, and Google Drive was used to manage the report.

C++ syntax and naming conventions have been used during development. Coding standards can be found here: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>

To aid further development of our SPA, description of functions are written as comments in the header files of our components.

5 Testing

Visual Studio was used to perform unit testing, and AutoTester was used to run our program as a system and test its workability.

Testing was done in 3 stages - unit, integration, and finally system. Adopting good programming habits, unit testing was done alongside development of the component. Integration testing was done to ensure information could be parsed from one component to the next through implemented APIs, and lastly system testing was done to ensure a fully workable SPA system.

5.1 Unit Testing

Unit testing was done on the 4 individual components during development. The components were individually scrutinised for functionality by writing test cases; these test cases were automated (using Visual Studio) and later used for regression testing when more methods were added to the component.

5.1.1 Parser

Parser Unit Test - Assign Statement Validation

```
TEST_METHOD(assignParse) {
    string input = "x=1+y;";
    Assert::IsTrue(isAssignStatement(input));
    input = "xyz=1+y;";
    Assert::IsTrue(isAssignStatement(input));
    input = "x = 1 + z ;";
    Assert::IsTrue(isAssignStatement(input));
    input = "xyz = 1 + z ; }";
    Assert::IsTrue(isAssignStatement(input));
    input = "x=1; 2";
    Assert::IsFalse(isAssignStatement(input));
    input = "=1;";
    Assert::IsFalse(isAssignStatement(input));
    input = " a = f +s = 3;";
    Assert::IsFalse(isAssignStatement(input));
}
```

Test purpose: To test the parser's ability to validate assign statements parsed through it

Test input: SIMPLE assign statements

Expected result: The assertions in the unit test; true if the assign statement is valid, and false otherwise.

5.1.2 PKB

PKB Unit Test – Modifies Updates Containers

```
TEST_METHOD(setModifiesCorrectly_wParent) { //set modifies for stmt and parent correctly

    Modify modify;
    vector<int> procCalledBy = {};
    vector<int> procCalls = {};
    vector<int> procCalledByStmt = {};

    vector<int> parentStar;
    parentStar.push_back(1);

    modify.setModifies(2, 2, parentStar);
    Assert::AreEqual(2, modify.getModifies(1)[0]); //parent 1 modifies var 2
    Assert::AreEqual(2, modify.getModifies(2)[0]); //stmt 2 modifies var 2
    Assert::AreEqual(1, modify.getModifiedBy(2)[0]); //var 2 modified by parent 1
    Assert::AreEqual(2, modify.getModifiedBy(2)[1]); //var 2 modified by stmt 2

    modify.setProcModifies(1, 2);
    Assert::AreEqual(modify.getProcModifies(1)[0], 2);
    Assert::AreEqual(modify.getProcModifiedBy(2)[0], 1);

    modify.setProcModifies(1, 1);
    Assert::IsTrue(modify.getProcModifies(1) == vector<int>{1, 2});
    Assert::AreEqual(modify.getProcModifiedBy(1)[0], 1);
}
```

Test purpose: To test if modifications in container statements will update “Modifies” table of parent statement.

Test input: SIMPLE container source code

Expected result: Variables modified by container statements should update modifies table of parent statement. In this case, parentStar has a vector<int> containing statement 1; this is the parent of statement 2 (i.e. first argument of the setModifies(stmt Num, var, parentStar) function). getModifies(1) (i.e. getModifies of statement 1) should return 2 (i.e. variable with index ID 2).

PKB Unit Test – Parent Set Correctly

```
TEST_METHOD(setParentCorrectly) {
    Parent parent;
    parent.setParent(1, 2);
    Assert::AreEqual(2, parent.getChild(1)[0]);
    Assert::AreEqual(1, parent.getParent(2)[0]);

    parent.setParent(2, 3);
    vector<int> actual = {2, 3};
    Assert::IsTrue(actual == parent.getChildStar(1));
    Assert::IsTrue(parent.getParentStar(3) == vector<int>{2, 1});

    parent.setParent(1, 4);
    Assert::IsTrue(parent.getChild(1) == vector<int>{2, 4});
}
```

Test purpose: To test if parent of container statements are set correctly.

Test input: SIMPLE container source code

Expected result: Variables in container statements should return their parent when getParent(child) is called. In this case, statement 1 can be seen to be the parent of statements 2 and 4.

5.1.3 Preprocessor

Preprocessor Unit Test – Correct Creation of queryStatement Object

```
QueryValidator queryValidator;
string query;
QueryElement selectQueryElement, suchThatQueryElement, patternQueryElement, withQueryElement;
QueryStatement expectedQueryStatement;
QueryStatement queryStatement;

query = "stmt s; assign a; procedure p; while w; if i; call c; variable v; Select BOOLEAN such that Modifies(3, \"a\") and Modifies(a, v)\"
      \"and Modifies(\"x\", \"x\") and Modifies(s, v) such that Modifies(w, v) and Modifies(i, v) and Modifies(p, v)";
Assert::IsTrue(queryValidator.parseInput(query));
selectQueryElement = QueryElement("empty", "empty", "BOOLEAN", "empty");
expectedQueryStatement.addSelectQuery(selectQueryElement);
suchThatQueryElement = QueryElement("3", "number", "empty", "a", "variable", "empty", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("a", "synonym", "assign", "v", "synonym", "variable", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("x", "variable", "empty", "x", "variable", "empty", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("s", "synonym", "stmt", "v", "synonym", "variable", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("w", "synonym", "while", "v", "synonym", "variable", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("i", "synonym", "if", "v", "synonym", "variable", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
suchThatQueryElement = QueryElement("p", "synonym", "procedure", "v", "synonym", "variable", "Modifies", "suchThat");
expectedQueryStatement.addNormalQueryElement(suchThatQueryElement);
queryStatement = queryValidator.getQueryStatement();
```

Test purpose: To test if parsed query creates the correct query elements and forms a queryStatement object.

Test input: SIMPLE query

Expected result: The query should be broken down into clauses and create multiple queryElements. In this case, we see a queryElement created for the BOOLEAN clause, one for “such that” Modifies, another for “and”, and so on. Once the query has been broken down to its elements, it is finally connected to form a queryStatement.

5.1.4 Evaluator

Evaluator Unit Test – Check Intermediate Tables Merging

```
TEST_METHOD(twoSynonymSameTable) {
    PKB pkb;
    QueryAnalyzer qa;
    QueryStatement qs;
    tuple<bool, vector<vector<string>>> clauseResult;
    pkb.setStatementType(1, "while");
    pkb.setStatementType(2, "assign");
    pkb.setStatementType(3, "assign");
    pkb.setParent(1, 3);
    pkb.setParent(1, 2);
    pkb.setFollows(2, 3);
    qa.setPKB(pkb);
    vector<vector<string>> result;
    vector<vector<string>> hardcode;
    vector<vector<vector<string>>> database;
    //intint

    vector<vector<string>> a{ {"1", "2", "3", "a"}, {"7", "8", "9", "b"} };
    vector<vector<string>> b{ {"2", "3", "a"}, {"8", "9", "b"} };
    qa.insertClauseResults(a);
    qa.restrictTableValues(a, 0, 1, 0, b);
    hardcode = { { "2", "3", "a" }, { "8", "9", "b" } };
    database = qa.getMergedQueryTable();
    for (int i = 0; i < database[0].size(); i++)
        for (int j = 0; j < database[0][i].size(); j++) {
            Assert::AreEqual(hardcode[i][j], database[0][i][j]);
        }
}
```

Test purpose: To test if merging algorithm for intermediate tables are correct.

Test input: Intermediate result vectors

Expected result: Vectors should correspondingly merge, and contain only variables seen in both vectors. In this case, vectors {"1", "2", "3", "a"} and {"2", "3", "a"} should merge to result in a vector {"2", "3", "a"};

5.2 Integration Testing

5.2.1 Front-End Integration Test

Front-End Test was conducted to test two aspects – (1) correct parsing, and (2) accurate population of PKB. To test this, all getter APIs provided by the PKB were used.

Front-End Test – Integration between Parser and PKB

```
TEST_METHOD(FrontEndTest) {
    PKB pkb;
    string filename = "..\\..\\..\\Tests09\\Sample-Source(actual).txt";
    Assert::IsTrue(Parse(filename, pkb));
    Assert::IsTrue(pkb.getAllCalls() == vector<string>{"q", "p"});
    Assert::IsTrue(pkb.getAllConstants() == vector<string>{"1", "2", "3", "5"});
    Assert::IsTrue(pkb.getAllProcedures() == vector<string>{"Example", "p", "q"});
    Assert::IsTrue(pkb.getAllStmt() == vector<int>{4, 14, 1, 2, 3, 5, 7, 8, 9, 11, 15, 17, 18, 19, 20, 21, 23, 24, 6, 13, 22, 10, 12, 16});
    Assert::IsTrue(pkb.getAllVariables() == vector<string>{"i", "x", "y", "z"});
    Assert::IsTrue(pkb.getAssign() == vector<int>{1, 2, 3, 5, 7, 8, 9, 11, 15, 17, 18, 19, 20, 21, 23, 24});
    Assert::IsTrue(pkb.getCall() == vector<int>{10, 12, 16});
    Assert::IsTrue(pkb.getCalledBy("p") == vector<string>{"Example"});
    Assert::IsTrue(pkb.getCalledByStar("q") == vector<string>{"Example", "p"});
    Assert::IsTrue(pkb.getCalls("p") == vector<string>{"q"});
    Assert::IsTrue(pkb.getCallsStar("Example") == vector<string>{"q", "p"});
    Assert::IsTrue(pkb.getChild(13) == vector<int>{14, 18, 19, 20});
    Assert::IsTrue(pkb.getChildStar(4) == vector<int>{5, 6, 7, 8, 9, 10, 11});
    Assert::IsTrue(pkb.getFollowedBy(12) == vector<int>{4});
    Assert::IsTrue(pkb.getFollowedByStar(21) == vector<int>{13});
    Assert::IsTrue(pkb.getFollows(13) == vector<int>{21});
    Assert::IsTrue(pkb.getFollowsStar(1) == vector<int>{2, 3, 4, 12});
    Assert::IsTrue(pkb.getIf() == vector<int>{6, 13, 22});
    Assert::IsTrue(pkb.getLastLine("q") == 24);
    Assert::IsTrue(pkb.getModifiedBy("x") == vector<int>{1, 4, 5, 10, 12, 13, 14, 15, 16, 18, 22, 24});
    Assert::IsTrue(pkb.getModifies(13) == vector<string>{"x", "z", "i"});
    Assert::IsTrue(pkb.getParent(7) == vector<int>{6});
    Assert::IsTrue(pkb.getParentStar(8) == vector<int>{6, 4});
    Assert::IsTrue(pkb.getPatternVariable("i") == tuple<vector<int>, vector<string>> { {3, 11, 17}, {"(5)", "(i)(1)-", "(i)(1)-"} });
    Assert::IsTrue(pkb.getProcModifiedBy("y") == vector<string>{"Example"});
    Assert::IsTrue(pkb.getProcModifies("q") == vector<string>{"x", "z"});
    Assert::IsTrue(pkb.getProcUsedBy("x") == vector<string>{"Example", "q", "n"});
}
```

Test Purpose: To check if parser is calling the right setters, populating the PKB accurately

Test Input: 1 Sample Source Code (downloaded from IVLE)

Test Result: Assert values of all getters provided by the PKB to check if relationship tables were populated correctly

5.2.2 Query Preprocessor Test

Validation of queries is split into 4 parts - (1) testQuerySuchthat, (2) testQueryWith, (3) testQueryPattern and (4) testQueryAll.

Query Validation - testQueryPattern

```
query = "variable v1,v#; assign a1,a#; Select v1 pattern a#(v#,_\"x+y\"_)| pattern a1(1, \"(x+y)\")pattern a1(v1, \"x\")";  
Assert::IsTrue(queryValidator.parseInput(query));  
  
query = "variable v1,v#; assign a1,a#; constant d; while w1, w2; Select v1 pattern w1(\"x\", _);  
Assert::IsTrue(queryValidator.parseInput(query));  
  
query = "variable v1,v#; if ifs1; constant d; while w1, w2; Select v1 pattern ifs1(\"xor \", _, _);  
Assert::IsTrue(queryValidator.parseInput(query));
```

Test purpose: To check “pattern” clauses with valid / invalid declarations and syntax

Test input: Queries with assign / while / if patterns; can contain multiple clauses

Test result: Returns true if query is valid, otherwise false

Query Validation - testQueryAll

```
query = "variable v1,v#; assign a1,a#; constant d; while w1, w2; Select v1 such that Modifies(6,\"x\") pattern a(_, _\"x\")";  
Assert::IsFalse(queryValidator.parseInput(query));  
  
query = "variable v1,v#; assign a1,a#; constant d; while w1, w2; Select v1 such that Modifies(6,\"x\") and Parent(1, _)pattern a1(v1, \"x\") with d.value = 4";  
Assert::IsTrue(queryValidator.parseInput(query));  
  
query = "variable v1,v#; assign a1,a#; constant d; while w1, w2; Select v1 pattern a#(v#,_\"x+y\"_) such that Parent(1, _)pattern a1(v1, \"x\")";  
Assert::IsTrue(queryValidator.parseInput(query));
```

Test purpose: To check queries with multiple and varying valid / invalid clauses, with valid / invalid declarations.

Test input: Query that expects boolean results

Test result: Returns true if query is valid, otherwise false

5.3 System (Validation) Testing

System testing was done using AutoTester.

5.3.1 Source 6

Description of source: A source code with less than a 100 lines of SIMPLE language code

Purpose of source: To test if Parser and PKB are able to handle a small amount of tables that can respond to basic queries

Description of queries: Most are SIMPLE queries (either with a single clause, or a small number of multiple clauses) to test that the system is able to handle a specific type of query (e.g. Follows / Affects*)

Purpose of queries: To test the ability to handle an adequate amount of data in the evaluator.

Number of queries (in queries-6) used to test specific types of queries:			
Follows: 15	Follows*: 14	Parent: 6	Parent*: 6
Modifies: 7	Uses: 5	Calls: 9	Calls*: 8
Next: 30	Next*: 28	Affects: 9	Affects*: 5
Pattern: 25	With: 23		
Mixed clauses (including stress) : 27		Mixed clauses tuple (including stress): 38	
Invalid / Meaningless : 34		BOOLEAN: 16	

The following are some queries that are taken out of queries 6.

	Example Queries	Purpose
Follows	1) assign a; Select a such that Follows(2,3) 2) assign a; Select a such that Follows(a,22) 3) assign a; Select a such that Follows(22,a) 4) assign a; if ifs; Select a such that Follows(ifs, a) 5) assign a; while w; Select w such that Follows(w, a) 6) stmt s; while w; Select s such that Follows(s,w)	(1) Tests Follows is functional when no synonym is involved (2) and (3) tests Follows is functional when 1 synonym involved (4), (5) and (6) tests Follows is functional when 2 synonym are involved (7) and (8) tests Follows is

	<p>7) stmt s; while w; Select w such that Follows(w,_)</p> <p>8) stmt s; if ifs; Select ifs such that Follows(_, ifs)</p>	functional when underscore / wildcard is involved.
Modifies	<p>1) assign a; variable v; Select a such that Modifies(a, _)</p> <p>2) assign a; variable v; Select a such that Modifies(a, "hOne")</p> <p>3) procedure p; Select p such that Modifies(p, "hOne")</p> <p>4) variable v; Select v such that Modifies("SystemTestThree", v)</p>	<p>(1) Tests Modifies is functional when a synonym and underscore / wildcard are involved</p> <p>(2) Tests if stringLiteral can be taken in as an argument 2</p> <p>(3) and (4) tests the case for when a procedure is in argument 1, it can be either a synonym or a stringLiteral</p>
With Clauses	<p>1) stmt s; Select s with s.stmt# = 14</p> <p>2) if ifs; Select ifs with ifs.stmt# =14</p> <p>3) assign a; Select a with 15 = a.stmt#</p> <p>4) while w; Select w with w.stmt# = 23</p> <p>5) call c; Select c with c.stmt# = 14</p> <p>6) procedure p; Select p with p.procName = "blah"</p> <p>7) call c; Select c with "blah" = c.procName</p> <p>8) prog_line n; Select BOOLEAN with n = 62</p> <p>9) variable v; Select v with v.varName = "SystemTestFour"</p>	<p>(1), (2), (3), (4) and (5) tests stmt# is functional for the different stmt types (i.e. assign / call / while / if)</p> <p>(6) and (7) tests procName is functional for different procName (i.e. procedure / call)</p> <p>(8) tests synonym(prog_line) is functional for only integer types (e.g. 62)</p> <p>(9) tests varName is functional for variable type</p>

Pattern Clauses	<ol style="list-style-type: none"> 1) assign a1; Select BOOLEAN pattern a1(, "hOne+hTwo") 2) assign a1; variable v; Select a1 pattern a1(v, _"kThree*kFour"_) 3) assign a1; variable v; Select a1 pattern a1(, _) 4) assign a1; variable v; Select a1 pattern a1("hFour", _) 5) assign a1; variable v; Select a1 pattern a1("hFour", "hOne+hTwo*hThree") 6) assign a1; variable v; Select a1 pattern a1(v, "((hOne+(hTwo*hThree)))") 7) if ifs; variable v; Select ifs pattern ifs(, _ , _) 8) if ifs; variable v; Select ifs pattern ifs(v, _ , _) 9) if ifs; variable v; Select ifs pattern ifs("hOne", _ , _) 10) while w; variable v; Select w pattern w(, _) 11) while w; variable v; Select v pattern w(v, _) 12) while w; variable v; Select BOOLEAN pattern w("aOne", _) 	<p>(1), (2) and (3) tests argument 2 of assign pattern clauses to be an exact string (i.e. "hOne + hTwo"), subString (i.e. _"kThree*kFour"_), or underscore / wildcard (i.e. '_')</p> <p>(1), (3), (4) and (5) tests argument 1 of assign pattern clauses to be an exact string (i.e. "hFour"), synonym(variable) (i.e. v) or underscore / wildcard (i.e. '_')</p> <p>(6) specifically tests argument 2 of assign pattern clauses for a well-formed expressions as defined by SPA grammar rules</p> <p>(7), (8) and (9) tests argument 1 of if pattern clauses to be an underscore / wildcard (i.e. '_'), exact string (i.e. "hFour") or synonym(variable) (i.e. v)</p> <p>(10), (11) and (12) tests argument 1 of while pattern clauses to be an underscore/wildcard (i.e. '_'), exact string (i.e. "hFour") or synonym(variable) (i.e. v)</p>
Invalid Queries	<ol style="list-style-type: none"> 1) assign a; variable v; Select a pattern a(v, "(aOne") 2) if ifs; variable v; Select ifs pattern ifs(, 2, _) 3) assign a; variable v; Select a pattern a(v, "(aOne")) 4) stmt s; Select BOOLEAN with s.stmt# = "string" 	<p>(1) and (3) are examples of invalid pattern queries with invalid syntax, which will be caught by queryValidator</p> <p>(2) is an example of invalid pattern queries with invalid arguments, which will be caught by</p>

	<p>5) constant c; Select c with c.value = "string"</p> <p>6) procedure p1, p2; Select p1 such that Calls*("SystemTestOne", "SystemTestOne")</p> <p>7) assign a; Select a such that Next(2, "string")</p>	<p>queryValidator</p> <p>(4) and (5) are examples of invalid with queries with invalid arguments, which will be caught by queryValidator</p> <p>(6) is an example of invalid suchThat queries with invalid arguments; in this case, as procedures are not allowed to recurse, this will be caught by queryValidator</p> <p>(7) is an example of invalid suchThat queries; in this case, as Next arguments are not allowed to have a string, this will be caught by queryValidator</p>
Mix Queries	<p>1) variable v; assign a1, a2; if ifs; procedure p; stmt s; while w; Select v such that Modifies(p, v) and Uses(s, v) such that Parent*(ifs, s) and Follows*(s, a1)</p> <p>2) assign a; if ifs; variable v1, v2; procedure p1; Select <a, ifs> such that Parent*(ifs, a) and Modifies(a, v1)</p> <p>3) while w; if ifs1, ifs2; assign a1, a2; stmt s; constant c; variable v; Select <ifs1, s, a1> such that Affects*(s, a1) with s.stmt# = 3 such that Next*(s, ifs2) and Parent(ifs1, ifs2) pattern ifs2(v, _) with v.varName = "bOne" pattern a1("dOne", _)</p>	<p>(1) tests queryValidator and queryEvaluator are able to handle multiple "such that" queries with 1 synonym (i.e. v) as a select query, while obtaining correct information from the PKB</p> <p>(2) tests queryValidator and queryEvaluator are able to handle multiple "such that" queries with tuples while obtaining correct information from the PKB</p> <p>(3) tests queryValidator and queryEvaluator are able to handle all types of multiple queries with tuples while obtaining correct</p>

	4) assign a1, a2, a3, a4, a5, a6, a7, a8, a9; if ifs1, ifs2, ifs3, ifs4; while w1, w2, w3; Select a5 such that Next(a1,a2) and Next(a2, a3) and Next(a3, a4) and Next(a4, a5) and Next(a5, w1) and Next(w1, a6) and Next(a6, w2) and Next(w2, a7) and Next(a7, a8) and Next(a8, a9) and Next(a9, ifs1)	information from PKB (4) tests queryEvaluator are able to handle stress tests while obtaining correct information from PKB
--	---	---

Table 5.3.1.1

As seen in source 6, the main purpose of our queries are to check that the our SPA is able to run smoothly with all types of permitted arguments so that we need not check for all possible combinations of arguments again when dealing with (more) multiple clauses. Some form of stress test is also conducted so that we know that our SPA is able to handle a query with many clause.

5.3.2 Source 7

Description of Source 7: A source code with less than 200 line of SIMPLE language code

Purpose of Source 7: To test if Parser and PKB are able to handle an adequate amount of tables (i.e. = an extended and longer version of source 6)

Description of queries 7: Most are simple queries (i.e. either a single clause or a small number of multiple clauses) to test that the system is able to handle specific types of queries (e.g. Follows / Affects*)

Purpose of queries 7: To test the ability to handle an adequate amount of data in the evaluator. Invalid queries are lessened as most as tested by queries 6.

Number of queries (in queries-7) used to test specific types of queries:

Follows: 5	Follows*: 5	Parent: 5	Parent*: 5
Modifies: 7	Uses: 6	Calls: 6	Calls*: 4
Next: 7	Next*: 7	Affects: 8	Affects*: 8
Pattern: 12	With: 9	Invalid: 1	

	Example Queries	Purpose
Uses	1) Select BOOLEAN such that Uses("rain", "timeToSleep") 2) call c; variable v; Select v such that Modifies(c, v)	(1) tests if queryValidator is able to take in strings for both arg1 and arg2, if queryEvaluator is able to recognise if arg1 is a procedureName and arg2 is a variableName, and tests if the correct information (e.g. getUses) is obtained from PKB (2) test if queryValidator is able to take in arg1 as a Call design entity, if queryEvaluator is able to recognise arg1 as a Call design entity and arg2 as a variable synonym, and tests if the correct information (e.g. getUses) is obtained from PKB
Mix	1) while w; if ifs; assign a1, a2; stmt s; variable v1, v2; pocedure p; Select <w, v2> such that Parent*(w, ifs) and Modifies(ifs, v1) pattern a1(v1, _"airconNotGood"_) and a2(v2, _) with v2.varName = p.procName	(1) tests if queryValidator and queryEvaluator are able to handle multiple valid clauses while PKB is able to parse and store correctly relationships of substantially long SIMPLE code

Table 5.3.2.1

From source 6 / 7 to source 8, to reduce the time and amount of effort, and yet having sufficient test cases for system testing, we have applied the concept of equivalence partitioning. For example, with reference to Table 5.3.1.1, for Follows queries, all possible permutations of Follows queries in terms of number synonyms were tested and verified correct. As such, extensive testing for Follows queries in queries-8 of source 8 is skipped. Focus is shifted to more complicated queries in source 8, such as those which are more likely to timeout.

5.3.3 Source 8

Description of source: A source code with less than 500 lines of SIMPLE language code

Purpose of source: To test if Parser and PKB are able to handle large numbers of tables, trying to simulate a situation close to the actual testing.

Description of queries: Test a small amount of single clause query, and a mix of multiple queries

Purpose of queries: To test the ability to handle an adequate amount of data in the evaluator. Invalid queries have been excluded as the main purpose of these queries are to test the ability to handle large amounts of data

Number of queries (in queries-8) used to test specific types of queries:

Follows*: 1	Parent*: 1	Modifies: 1	Uses: 1	Next: 1	Next*: 1
Affects: 3	Affects: 6	Mixed (including stress): 12			

	Example Queries	Purpose
stmtLst	1) stmtLst s; Select s	(1) tests if the implementation of stmtLst is functional in both the PQL and PKB
Mix/Complex Queries	1) stmt s1,s2; call c; procedure p; Select <c,p> with p.procName = c.procName such that Affects*(s1,s2) such that Follows*(s2,c) 2) stmt s1,s2; call c; procedure p; Select <c.procName,p.procName> such that Affects*(s1,s2) such that Follows*(s2,c) with p.procName = c.procName 3) assign a; variable v1,v2; stmt s1; while w; prog_line n; if ifs; procedure p; Select <ifs, a> such that Affects(s1, a) pattern ifs("China", _ , _) and	(1) and (2) are mixed queries testing relationships that are likely to timeout (e.g. Affects*), combined with other clauses. This tests if the evaluator is able to obtain the correct information from the PKB, and perform table merging effectively and efficiently (3) is a mixture of complex queries focusing on testing merging of tables by the evaluator, and tests the

	$a(v1, _ "expenses" _)$ such that $Next^*(s1, n)$ and $Modifies(p, v1)$ and $Parent^*(ifs, a)$ pattern $w(_, _)$ with $a.stmt\# = n$ with $v1.varName$ $= "China"$	information obtained from PKB is correct (in this case timeouts are less likely to occur)
--	--	--

Table 5.3.3.1

With System testing on Source 8, we are able to confirm that our SPA is indeed able to run smoothly on a large SIMPLE source code that is able to handle multiple clause queries.

5.3.4 Regression Testing

Regression Testing is also another technique that we employ as part of our system testing. For example, during week 11 where $Affects^*$ is not implemented. Source 6 was written with the intent to Testing all clauses except $Affects^*$. After $Affects^*$ has been implemented, we ran through Source 6 with its corresponding queries again to ensure that the queries that was once giving correct expected results continue to give the correct ones. Another example will be the introduction of query optimization, where we run through the old set of queries again and ensure that we continue to obtain the expected results.

5.3.5 AutoTester

Auto-testing is a powerful tool that aids with the running of a large number of queries quickly, and viewing the output of these queries systematically on the command prompt or an output xml file (which can be viewed using firefox). This allowed for viewing and comparing of the expected and actual results in a single glance, and identification of queries giving unexpected results easily, as opposed to using visual studio which requires manual output of our actual results. The following are some snippets of how Auto-tester was used in our system testing.

```
D:\School\Project\Code09\Release>AutoTester D:\School\Project\Tests09\Sample-Source-6.txt D:\School\Project\Tests09\Sample-Queries-6.txt queries6.xml
```

Figure 5.3.4.1

Summary		
AVE-PERF:		Average Performance computed only for test cases that passed
Total-TC:	313	Total number of test cases run by Autotester (including CRASH)
TE:	9	Total number of test cases that failed (WR+TME+CRASH+EXCEPT)
PASS:	304	Total number of test cases that produced correct result (TOTAL-TC-TE)
WR:	7	Total number of tests that run ok but produced wrong results
List WR test case numbers: [186] [187] [303] [304] [311] [316] [317]		
UE:		Total number of unique errors, i.e., groups of test cases (WR+TME+CRASH+EXCEPT) that most likely failed for the same reason; entered manually after analysis
Comments: Here we enter comments during presentation		
TME:	2	Total number of tests that timeout in extended time
List TME test case numbers: [180] [188]		
CRASH:	0	Total number of tests that crashed
List CRASH test case numbers:		
EXCEPT:	0	Total number of tests that raised exception
List EXCEPT test case numbers:		

Figure 5.3.4.2

Hence, with a better view on what went wrong in our system, it was easy to focus on queries that returned wrong / expected result, saving time required to narrow down queries.

6 Discussion

Our team faced challenges completing the project; with 4 members each of us had to take on an additional 10 hours of development. Completing the development of the SPA system with each member being solely responsible for a component. There is definite understanding of the other components by all team members, but the implementation of each component is usually handled alone as everyone has many tasks to attend to. There is yet integration testing, writing of test cases and documentation to attend to, and all these tasks are slightly overwhelming when we are facing troubles with our component implementations.

7 Documentation of Abstract APIs

7.1 PKB

<code>vector<string> getAllVariables()</code> Returns vector string of all variables in PKB
<code>vector<string> getAllConstants()</code> Returns vector string of all constants in PKB
<code>getAllProcedures()</code> Returns vector string of all procedures in PKB
<code>setFollows(int statementNum1, int statementNum2)</code> Returns void
<code>vector<int> getFollows(int statementNum)</code> Returns a vector containing one int value, the statement number which follows the input statement
<code>getFollowedBy(int statementNum)</code> Returns a vector containing one int value, the statement number which is followed by the input statement
<code>vector<int> getFollowsStar(int statementNum)</code> Returns a vector containing integers, the statement numbers which follows* the input statement
<code>vector<int> getFollowedByStar(int statementNum)</code> Returns a vector containing integers, the statement numbers is followed* by the input statement

void setParent(int statementNum1, int statementNum2)
Returns void
vector<int> getParent(int statementNum)
Returns a vector containing one int value, the statement number which is the parent of the input statement
vector<int> getChild(int statementNum)
Returns a vector containing one int value, the statement number which is the child of the input statement
vector<int> getParentStar(int statementNum)
Returns a vector containing integers, the statement numbers which are the parent* of the input statement
vector<int> getChildStar(int statementNum)
Returns a vector containing integers, the statement numbers which are the child* of the input statement
void setModifies(int statementNum, string varName)
Returns void
void setProcModifies(string procName, string varName)
Returns void
vector<string> getModifies(int statementNum)
Returns a vector containing strings the names of the variables the input statement modifies
vector<int> getModifiedBy(string varName)
Returns a vector containing integers of statement numbers that modified the variable varName
vector<string> getProcModifies(string procName)
Returns a vector containing strings the names of the variables the input statement modifies
vector<string> getProcModifiedBy(string varName)
Returns a vector containing strings of procedure names that modified the variable varName
void setUses(int statementNum, string varName)
Returns void
void setProcUses(string procName, string varName)
Returns void
vector<string> getUses(int statementNum)
Returns a vector containing strings the names of the variables the input statement uses
vector<int> getUsedBy(string varName)
Returns a vector containing integers of statement numbers that used the variable varName
vector<string> getProcUses(string procName)
Returns a vector containing strings the names of the variables the input statement uses

<code>vector<string> getProcUsedBy(string varName)</code>
Returns a vector containing strings of procedure names that used the variable <code>varName</code>
<code>void setCalls(int statementNum, string procName1, string procName2)</code>
Returns void
<code>vector<string> getCalls(string procName)</code>
Returns a vector containing strings the names of the procedures it directly calls
<code>vector<string> getCalledBy(string procName)</code>
Returns a vector containing strings the names of the procedures the input procedure is called by
<code>vector<string> getCallsStar(string procName)</code>
Returns a vector containing strings the names of the procedures it calls
<code>vector<string> getCalledByStar(string procName)</code>
Returns a vector containing strings the names of the procedures the input procedure is called by

7.2 Preprocessor

7.2.1 SynonymEntityPair

<code>vector<string> getSynonymList()</code>
Return a vector<string> of synonyms is associated with a particular entity
<code>string getEntity()</code>
Returns a string of the entity that is in the SynonymAndEntityPair
<code>SynonymEntityPair(string, vector<string>)</code>
Creates a synonym entity pair.

7.2.2 QueryValidator

<code>bool parseInput(string str)</code>
Takes in a string argument i.e. query. Returns true if query is valid, else false
<code>bool isValidQueryLine(string selectString)</code>
Checks if input string (program line) is a valid query.
<code>bool isValidDeclarationRegex(String str)</code>
Takes in an argument string and checks if it matches the allowed grammar for declaration of synonyms. Returns true if valid, else false
<code>bool isValidSelectInitialRegex(string str)</code>

Checks if the result-cl is of permitted grammar. Returns true if valid, else returns false.
<p><code>bool isValidSuchThat(string str, string syn)</code></p> <p>Takes in 2 string arguments and check if the such that clause is valid. Returns true if valid, else false.</p>
<p><code>bool isValidSuchThatRegex(string str)</code></p> <p>Takes in a string arguments and checks if the string matches the grammar specified in handbook for suchthat-cl. Returns true if matches, else false.</p>
<p><code>bool isValidSuchThatExtendedRegex(string str)</code></p> <p>Takes in a string argument that may consists of multiple such that clauses and checks if this string matches suchthat-cl. Returns true if matches, else false.</p>
<p><code>bool isValidWithRegex(String str)</code></p> <p>Takes in a string argument that consists of with clauses and checks if this string matches the with-cl grammar. Returns true if matches, else false</p>
<p><code>bool isValidWithExtendedRegex(String str)</code></p> <p>Takes in a string argument that may consists of multiple with clauses and checks if this string matches with-cl. Returns true if matches, else false</p>
<p><code>bool isValidPattern(string str)</code></p> <p>Takes in a string argument that can consists of one or more pattern clause and checks if this string has fulfilled both the grammar rules and permitted arguments. Returns true if valid, else false</p>
<p><code>bool isValidPartialPatternRegex(string str)</code></p> <p>Takes in a string argument that checks from the start of string up to argument1 and see if it matches the permitted grammar rules. Returns true if matches, else false.</p>
<p><code>bool isValidWhilePatternRegex(string str)</code></p> <p>Takes in a string argument and checks if it matches the permitted grammar rules for a while type pattern argument. Returns true if matches, else false.</p>
<p><code>bool isValidIfPatternRegex(string str)</code></p> <p>Takes in a string argument and checks if it matches the permitted grammar rules for an if type pattern argument. Returns true if matches, else false.</p>
<p><code>bool isValidAssignPatternRegex(string str)</code></p> <p>Takes in a string argument and checks if it matches the permitted grammar rules for an assign type pattern argument. Returns true if matches, else false.</p>
<p><code>bool isEntityAndSynonym(string str)</code></p> <p>Takes in a string str argument which contains of the initial part of queried synonym. Returns true if the str is valid design entity, else false.</p>
<p><code>bool isValidEntity(string)</code></p> <p>Takes in a string argument and check if entity is valid. Returns true if valid, else false.</p>

bool isWildcard(string arg)
Checks if argument is a wildcard. Returns true if valid, else false.
bool isExactString(string arg2)
Checks if input argument is an exact string match for pattern. Returns true if valid, else false.
bool isValidExpr(string str)
Checks if input string is an 'expr' as specified in the grammar rules. Returns true if matches, else false
bool isValidExprUnder(string str)
Checks if input string is an 'expr' with additional underscores at both ends as specified in the grammar rules. Returns true if matches, else false.
vector<string> splitToSentences(string str)
Takes in a given string and split them according to the clauses type and stores them into vector. Returns this vector.
QueryStatement getQueryStatement()
Returns a QueryStatementObject

7.2.3 Relationship

vector<string> getArg1()
Returns the vector of design entities of the first argument of a relationship
vector<string> getArg2()
Returns the vector of design entities of the second argument of a relationship

7.2.4 RelationshipTable

bool isValidArg(string rel, string type, int numArg)
Takes in a relationship argument and the type of design entity the argument is, to be used for public calls, calls isValidArg1 or isValidArg2 accordingly. Returns true if the type of design entity exists in the relationshipTable, else false
bool isRelationshipExists(string rel)
Takes in a string argument rel and checks if this string is a key in the unordered_map(relationshipTable). Returns true it exists, else false.

7.2.5 QueryElement

string getSelectEntity()

Returns the entity associated with synonym being queried in Select query.
string getSelectSynonym() Returns the synonym bring queried in the Select query.
string getSelectType() Returns the type that is being queried in the Select query
string getSynAttr() Returns the string that is a concat of attrName or string 'synonym'
string getSuchThatRel() Returns the type of relationship
string getSuchThatArg1() Returns the first argument that is being queried on in a relationship
string getSuchThatArg1Type() Returns the first argument type that is being queried on in a relationship string
string getSuchThatArg1Entity() Returns the first argument design entity that is being queried on in a relationship string
getSuchThatArg2() Returns the second argument that is being queried on in a relationship
string getSuchThatArg2Type() Returns the second argument type that is being queried on in a relationship
string getSuchThatArg2Entity() Returns the second argument design entity that is being queried on in a relationship
string getPatternArg1() Returns the first argument of a pattern
string getPatternArg2() Returns the second argument of a pattern
string getPatternArg3() Returns the third argument of a pattern
string getPatternEntity() Returns the entity of the pattern
string getPatternSynonym() Returns the synonym of the pattern
string getPatternArg1Type() Returns the first argument type of a pattern
string getPatternArg2Type() Returns the second argument type of a pattern
string getPatternArg3Type() Returns the third argument type of a pattern

string getPatternArg1Ent()
Returns the first argument design entity of a pattern
string getClauseType()
Returns the type of clause the particular queryElement belongs to.

7.2.6 QueryStatement

vector<QueryElement> getSelectQueryElement()
Returns the vector<QueryElement> that is associated with Select.
vector<SynonymEntityPair> getSynonymEntityList()
Returns the synonym and Entities that were declared in a vector<SynonymEntityPair>
vector<QueryElement> getWithQueryElement()
Returns the vector<QueryElement> that is associated with 'with'
vector<QueryElement> getNormalQueryElement
Returns the vector<QueryElement> object that is associated with all such that clauses(excluding Next*/Affects/Affects*) and pattern clauses.
vector<QueryElement> getHardQueryElement
Returns the vector<QueryElement> object that is associated with all such that clauses(only Next*/Affects/Affects*)
multimap<string, pair<int, int>> getNormalMultiMap()
Returns a multimap<string,pair<int,int>> object which has a key that is a synonym that maps to value <argument number, index number in normalQueryElement>
multimap<string, pair<int, int>> getHardMultiMap()
Returns a multimap<string,pair<int,int>> object which has a key that is a synonym that maps to value <argument number, index number in hardQueryElement>
void setInvalidQueryBoolean()
Sets the private attribute 'resultBoolean' from false to true
void addNormalQueryElement(QueryElement)
Adds a queryElement of 'normal' type(Implies all such that and pattern clauses excluding Next*/Affects/Affects*) to the vector of normalQueryElements existing in a queryStatement object
void addNormalQueryElement(QueryElement)
Adds a queryElement of 'hard' type(Implies all Next*/Affects/Affects* clauses) to the vector of hardQueryElements existing in a queryStatement object
void addNormalMultiMap(string, int, int)
Adds a multiMap of 'normal' type(all such that and pattern clauses excluding Next*/Affects/Affects*) to the existing normalMultiMap in a queryStatement object.

void addHardMultiMap(string syn, int argumentNum, int idxAtHardQueryElements)
Adds a multiMap of 'hard' type(all Next*/Affects/Affects* clauses) to the existing hardMultiMap in a queryStatement object
int getNormalQueryElementsSize()
Returns the size of the private attribute normalQueryElements which is a vector
int getHardQueryElementsSize()
Returns the size of the private attribute hardQueryElements which is a vector
bool getInvalidQueryBoolean()
Returns the value of private attribute resultBoolean
void addSelectQuery(QueryElement)
Adds the select portion (Without clauses) of a query statement as a queryElement into the object
void addWithQuery(QueryElement)
Adds the with portion (without clauses) of a query statement as a queryElement into the object
void addSynonymEntityList(vector<synonymEntityPair>)
Adds the declared synonymEntityPair to QueryStatement Object.

7.3 Evaluator

7.3.1 queryAnalyzer

void QueryAnalyzerI()
Initialize the evaluator with empty mappings, PKB and Query Statement
vector<string> runQueryAnalyzerI()
Runs query Analyzer on query and returns a vector<string> type of the result.
void setPKB()
Inputs a read-only copy of the PKB into the query Analyzer