



NATIONAL UNIVERSITY OF SINGAPORE

CS3201/2 Iteration 2

16th October 2017

SEM 1, AY2017/2018

Submitted By :

Team 11

Consultation Hour: Tuesday, 1:00-2:00 pm

Team Member	E-mail	Phone number
Akankshita Dash	akankshita.dash@u.nus.edu	84010419
Chua Ping Chan	a0126623@u.nus.edu	90876406
Lau Wen Hao	a0121528@u.nus.edu	84991295
Marcus Ng	marcusngwenjian@u.nus.edu	97502493
Zhang Ying	e0006954@u.nus.edu	82289895
Zhuang Lei	e0003940@u.nus.edu	98374236

Contents

1	Scope of SPA Implementation	4
1.1	Parser	4
1.2	PKB	4
1.3	PQL	4
2	Development Plan	6
2.1	Development Plan for SPA	6
2.2	Development Plan for Iteration 2	8
3	SPA Design	9
3.1	Overview	9
3.2	Design of SPA Components	10
3.2.1	Parser	10
3.2.2	PKB	17
3.2.3	Design Extractor	24
3.2.4	PQLMain	29
3.2.5	Query Validator	29
3.2.6	Query Optimizer (plan)	34
3.2.7	Query Evaluator	36
3.2.8	Result Formatter	42
3.3	Design decisions	45
3.3.1	Parsing Strategy	45
3.3.2	PKB Design: Representation of Design Abstraction	46
3.3.3	PKB Design: Evaluating reverse relationships	47
3.3.4	PKB Design: CFG Design	48
3.3.5	Pattern Matching: Representation of expressions	49

3.3.6	Query Validation Design	50
3.3.7	Query Evaluation Design	52
3.3.8	Data Structures for Intermediate Result	53
4	Documentation and Coding standards	54
4.1	Abstract API and Concrete API	54
4.2	Naming Conventions	54
4.2.1	Formatting	55
4.3	Class hierarchy	55
5	Testing	55
5.1	Test Plan	56
5.1.1	Test Schedule for All Iterations	56
5.1.2	Test Objectives	57
5.2	Test Utilities	57
5.2.1	Test Utilities Class	57
5.2.2	Test Objectives	57
5.2.3	Script for automating System Testing	57
5.3	Unit Tests	60
5.3.1	PKB Unit Test	60
5.3.2	PQL Unit Test	60
5.4	Integration Tests	65
5.4.1	Parser - PKB SubSystem	65
5.4.2	PQL - PKB SubSystem	66
5.5	System Tests	66
6	Discussion	70
6.1	Technical Difficulties	70

6.2	Project Management	70
6.3	Workload and Time Management	70
7	Future Plan for Iteration 3	71
7.1	Features	71
Appendix A Documentation of Abstract APIs		71

1 Scope of SPA Implementation

Our prototype for SPA meets all the requirements for Iteration 2 and is a fully operational mini-SPA.

1.1 Parser

Our **SPA tool** is now able to parse if-else statements, call statements and procedures in addition to previous parsing of multiple statements, assignments and while loops.

1.2 PKB

PKB now supports **Calls, Calls*, Next, Next*, Pattern If, Pattern While** as well as **Modifies** and **Uses** for procedures, in addition to previous implementations of **Follows, Follows*, Parent, Parent*, Uses, Modifies** and **Pattern**

1.3 PQL

The QueryProcessor is able to evaluate PQL Queries with multiple **such that, pattern, and** and **with** clauses. It can also handle selection of **BOOLEAN** types.

2 Development Plan

2.1 Development Plan for SPA

	Tasks	Team Members					
		Andy	Zhang Ying	Ping Chan	Marcus	Dash	Wen Hao
Iteration 1	Design PKB API	Y	Y	Y			
	Build SIMPLE Parser			Y			
	Build PKB Tables	Y	Y				
	Build Design Extractor		Y				
	Build QueryValidator				Y	Y	Y
	Build QueryTree						Y
	Build PQLMain						Y
	Build QueryEvaluator				Y	Y	Y
	Perform Unit Testing	Y	Y	Y	Y	Y	
	Perform Integration Testing		Y	Y	Y	Y	
	Perform System Testing	Y	Y	Y	Y	Y	Y
	Write documentation	Y				Y	
Iteration 2	Extend Parser			Y			
	Extend Design Extractor		Y				
	Extend PKBTables	Y	Y				
	Implement Design Patterns			Y	Y		Y
	Extend QueryValidator				Y	Y	
	Change QueryTree						Y
	Extend QueryEvaluator					Y	Y
	Build ClauseResult			Y			
	Change PQLMain				Y	Y	
	Build ResultFormatter					Y	
	Perform Unit Testing		Y	Y	Y	Y	Y
	Perform Integration Testing		Y	Y	Y	Y	
	Perform System Testing	Y	Y	Y	Y	Y	Y
	Write documentation	Y	Y	Y	Y	Y	Y
Iteration 3	Extend Parser			Y			
	Extend Design Extractor	Y	Y				
	Extend PKBTables	Y	Y				
	Extend QueryValidator				Y		
	Extend QueryEvaluator						Y
	Build ClauseResult Handler			Y			
	Extend ResultFormatter					Y	
	Build Query Optimizer				Y	Y	
	Perform Unit Testing	Y	Y	Y	Y	Y	Y
	Perform Integration Testing	Y	Y	Y	Y	Y	Y
	Perform System Testing	Y	Y	Y	Y	Y	Y
	Write documentation	Y	Y	Y	Y	Y	Y

2.2 Development Plan for Iteration 2

Mini Iteration 1 - Week 6 to Recess Week				
PKB & Parser				
Activities	Ping Chan	Zhang Ying	Andy	-
Fixed bug found in Pattern Table			Y	
Implement Calls/Calls*		Y		
Parser - Parse Multiple Procedures	Y			
Parser - Add Calls Relations	Y			
PQL				
Activities	Marcus	Dash	Wen Hao	Ping Chan
Discuss design patterns	Y	Y	Y	Y
Refactor code	Y	Y	Y	
Extend validation for 'and'	Y			
Add support for Calls/Calls*			Y	
Add support for validation of with clause	Y			
Add support for Selection of BOOLEAN		Y		
Add support for validation of pattern	Y			
Intermediate Result (i.e. ClauseResult)				Y
Mini Iteration 2 - Week 7 to Week 8				
PKB and Parser				
Activities	Ping Chan	Zhang Ying	Andy	-
Implement pattern for while and if		Y	Y	
Refactor PKBMain		Y		
Add support for inter-procedural Uses/Modifies		Y		
Implement Next/Next*		Y		
Added support for if-else entity			Y	
Write API for PQL in PKB		Y		
Write API for Parser in PKB		Y		
Parser - Support If-Else Stmt	Y			
Parser - Support Call Stmt	Y			
Parser - Add Next Relations	Y			
Finish documentation for PKB component	Y	Y	Y	
Parser-PKB Integration Testing	Y			
PQL				
Activities	Marcus	Dash	Wen Hao	Ping Chan
Finish documentation for PQL component	Y	Y	Y	Y
Implement PQLMain	Y	Y		
Add support for Next/Next*			Y	
Write script to automate testing	Y			
Implement ResultFormatter		Y		

3 SPA Design

3.1 Overview

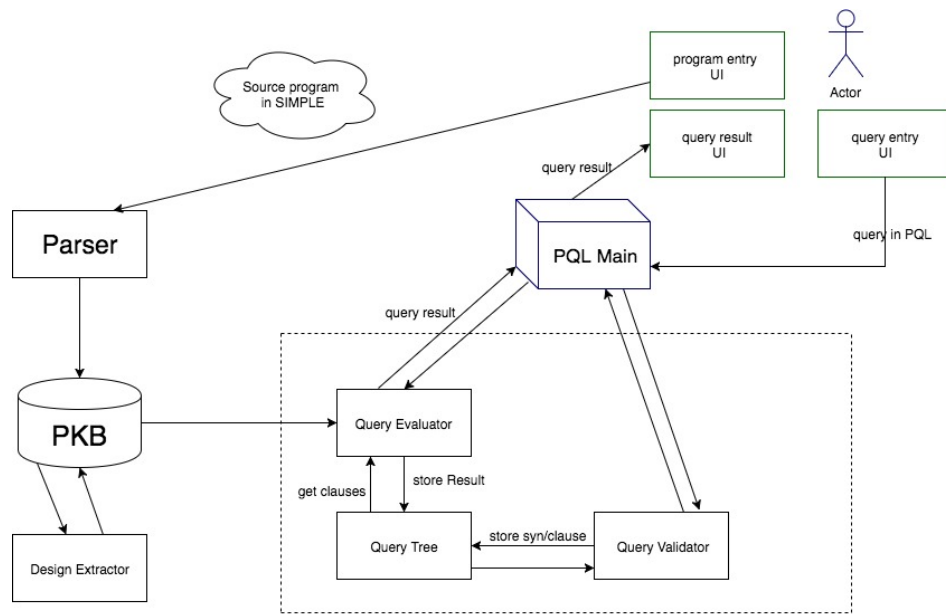


Figure 1: Main components of SPA

In SPA, there are two main components:

1. Program Knowledge Base (PKB)
2. Program Query Language (PQL)

Our SPA design is very similar to the original proposed design. In addition, to the proposed components, we have implemented a PQL Main, which controls the flow of all the other components in PQL and decides the sequence during query processing.

For iteration 3, we will implement Query Optimizer, which will come re-order clauses in order to optimize evaluation. It will work after Query Validator and before Query Evaluator.

3.2 Design of SPA Components

3.2.1 Parser

In SPA, Parser is responsible for parsing a given SIMPLE source code, validating the syntax, extracting information and storing it in PKB.

Whenever the Parser detects a syntax error, it will stop parsing the SIMPLE source and display an error feedback message to the user. It is only if the SIMPLE source code is syntactically correct, that the Parser will finish parsing and allow queries to be evaluated.

At the beginning of the parsing process, the Parser reads a given SIMPLE source file into a concatenated string, which is stored in-memory in a class attribute.

After the SIMPLE source content is extracted into a concatenated string, the Parser will make one parse just to ensure all the braces pairs up correctly. If there is a mismatch in the way the braces pair-up, the Parser will output a feedback message to indicate the syntax error and stop parsing at once.

If all the braces are paired up well, the Parser is now ready to Parse the SIMPLE source contents line by line. The Parser implemented is a predictive recursive descent parser, whereby the concatenated SOURCE string will be tokenized; the Parser starts by reading a token, and then determines what is the next token to expect.

Parser's Tokenizer

The tokenizer of the parser is made such that if given a SIMPLE program:

```
procedure First {  
  abc123 = 100 * x3;  
}
```

The tokenizer will break the string down to a list of tokens containing: 'procedure', 'First', '{', 'abc123', '=', '100', '*', 'x3', ';' and '}'. Any interleaving whitespace will be ignored.

For example, at the very beginning of the SIMPLE source, the Parser invokes a member function that parses procedure, and expects the upcoming token to be the keyword “procedure”. If the first token indeed matches the keyword, the Parser will move on to the next token and validate whether it is a valid procedure name or not, according to the SIMPLE grammar rules. If it is valid, the Parser will add the name of procedure into PKB. It will then expect an open brace character ‘{’ next, and so on.

Similar to parsing procedure, the Parser contains various methods to parse different entities such as statement lists, assignment statements, call statements, while statements and if-else statements. Each of these methods will ensure the SIMPLE source satisfies the grammar rules, e.g. correct operands, correct expressions in assignment statements, etc. If any syntax error is detected, Parser will throw an exception and stop parsing.

After the Parser finishes parsing a SIMPLE source, it signals to PKB that it’s finished, so that the Design Extractor within PKB starts computing transitive relations, e.g. Follows*, Parent*, Calls*, Next* and inter-procedural relations such as Uses and Modifies.

Uses/Modifies Relation: The Parser adds the Modifies and Uses relationship into PKB as and when the Parser parses the relevant statements. No additional data structures are required to store any information for adding Uses/Modifies relations.

Follows Relation: The Parser maintains a stack of statements in order to add Follows relations to PKB. Each inner stacks represents the statement list within a code block in the correct order, and the outer stack is used to store the information of nested statement lists due to container statements.

As each inner stack represents a statement list with statements in the correct order, at the end of a code block, Parser can pop the top-most statement list in the stack and process it by setting the Follows relation based on the sequence of statements in the code block.

Consider the following source code:

```
procedure First {  
1   i = 1;  
2   b = 200;  
3   c = a;  
4   while a {  
5       c = b - 5;  
6       b = c - a;  
7       x = x + 1;  
      }  
8   d = 300; }
```

The stack of statements stacks will be empty at first. While beginning to parse statement 1, since it is the first statement in a new statement list, an empty stack of statement will be created and pushed onto the stack of statements stacks. Statement 1, 2, 3 and 4 will then be pushed onto the inner stack in order. When parsing statement 5, as it is a new statement in a new nested statement list, another inner stack will be created. Statements 5, 6, 7 will then be added to this new inner stack and pushed into `__stackOfFollowsStack`. At the end of parsing statement 7, upon exiting the code block of the while statement, the top stack (marked in green) will be popped, and the follows relation among statements 5, 6 and 7 will be added to PKB. Then, statement 8 will be pushed to the blue statements stack. At the end of parsing statement 8, the end of procedure is reached, therefore the blue stack will be popped and the Follows relations amongst statements 1, 2, 3, 4 and 8 will be added to the PKB.

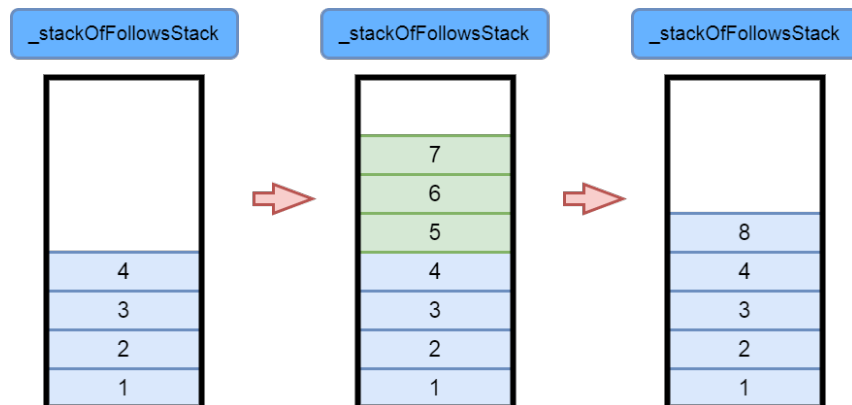


Figure 2: Keeping Track of Follows Relationship

Parent Relation: The Parser maintains a stack called `_parentStack` to keep track of the Parent relationship. Every time a parser encounters a container statement, that statement will be pushed into this stack. When exiting a container statement, the top most element of the stack will be popped. That way, while parsing the statements within a statement list of a nested container statement, the Parser will always be able to know all the Parents and transitive Parents (Parent*) while setting relations like Uses and Modifies in PKB. Consider the following source code:

```
[htbp] procedure First {
1   i = 1;
2   b = 200;
3   c = a;
4   while a {
5       c = b - 5;
6       b = c - a;
7       x = x + 1;
      }
8   d = 300; }
```

When the Parser parses statement 7, the state of the `_parentStack` would be as shown in

the figure below:

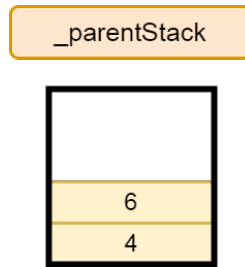


Figure 3: Keeping Track of Transitive Parents

Calls Relation : To add calls relation to PKB, the Parser will need to know the current procedure being parsed at all times. Hence, the Parser has a private attribute to store this information. When parsing a procedure call statement, the Parser validates the name of the procedure being called, and adds to PKB a call relation between the current procedure and the called procedure.

Next Relation : The Parser maintains a few data structures to add Next relation in PKB:

1. Two boolean variables to indicate whether the current statement is just outside a while statement or an if-else statement. These two variables are named `__justExitIfElseStmt` and `__justExitWhileStmt` respectively.
2. Two stacks to keep track of nested while statements and if-else statements respectively. These stacks are named `__whileStmtStack` and `__ifElseStmtStack` respectively.
3. An `unordered_map<int, pair<int, int> >` called `__ifElseStmtExitPoints` to store the mapping of if-else statements to the exit points of if-else statements.

```
procedure ABC {  
1   a = 1;  
2   b = 2;  
3   while x {  
4       c = 4;  
5       if x then {  
6           i = 6;  
           } else {  
7           j= 7;  
           }  
8       k = 8;  
       }  
9   m = 9;  
}
```

Whenever the Parser adds the Follows relation in PKB as described earlier, it also adds the Next relation.

When the Parser encounters an if-else statement, the statement number is pushed onto the `_ifElseStmtStack`. An integer variable is then used to store the if-else statement number for later use. When parsing the first statement in the if-block, the Parser will set Next relation for the if-else statement number and the first statement in the if-block. The Parser then continues to parse as usual until it reaches the last statement in the if-block. Now, the Parser uses another int variable to store the last statement in the if-block, then continues parsing until it reaches the first statement in the else-block. Here, the parser will set Next relation for the if-statement number and the first statement in the else-block. Before exiting the else-block, the Parser uses another int variable to store the last statement of the else-block, and push an element with the if-else statement number mapped to the last

statement of the if-block and else-block into `_ifElseStmtExitPoints`. The parser also sets `_justExitIfElseStmt` to true, and then exits the else block. After exiting the else-block, if there is a statement encountered, the Parser will refer to `_ifElseStmtExitPoints` to set Next relation for the statements:

1. The last statement of the if-block and the statement outside
2. The last statement of the else-block and the statement outside

The top element of the `_justExitIfElseStmt` and `_ifElseStmtStack` are then popped and the variable `_justExitIfElseStmt` is set back to false.

When encountering a while statement, the statement number will be pushed onto the `_whileStmtStack`. When parsing the first statement in the while-block, the Parser will set Next relation for the while statement number and the first statement in the while-block. The Parser then continues parsing until it reaches the last statement in the while-block. The Parser then add Next relation for the last statement in the while-block and the while statement. At this point, the `_justExitWhileStmt` variable is also set to true. After exiting the while-block, if there is a statement encountered, the Parser will refer to the `_whileStmtStack` to set the Next relation for the while statement and the statement just after the while-block. The top element of the `_whileStmtStack` is then popped and the boolean variable `_justExitWhileStmt` is set back to false.

For the above code, when parsing statement 8, the data structures that Parser maintains is shown in Figure 4.

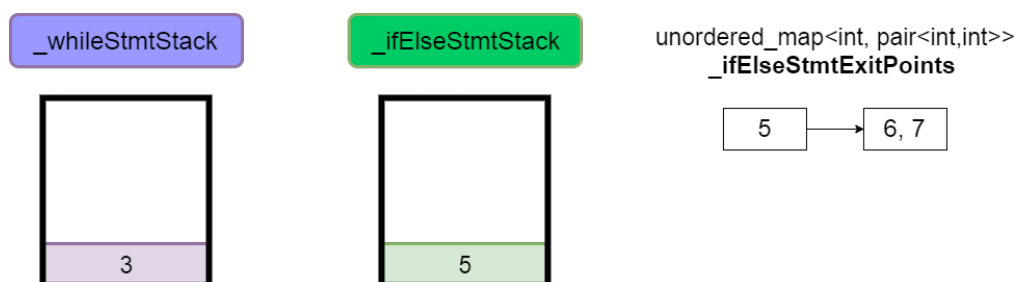


Figure 4: Keeping Track of Information of If-Else Statement to Add Next Relations

3.2.2 PKB

Program Knowledge Base (PKB) provides a storage to store program design abstractions such as Uses, Modifies and other relations. Table-driven approach is used for the design of the PKB structure. Not only does it provides a simple yet powerful way to achieve flexibility, but it also allow quick accessing of the data by the way of hash maps. It has been decided that different class table files are made according to the different design entities. This is to make extension to allow additions of new design entities easier and this could minimize modifications of code.

Tables vs AST

Abstract Syntax Tree (AST) is one of the conventional ways to store parsed data. However, when the QueryEvaluator accesses data, it will have to perform tree traversal, which could take a lot of runtime during query processing and is not an optimal way. Also, doing modifications to allow more design entities would be more complicated. Alternatively, hash maps can be used to store data. Having multiple hashmaps over AST allows faster searching, addition, deletion of data.

Hence, tables (in the form of hash maps) are used in this project so that query processing is done optimally in $O(1)$ time.

PKB contains the following components:

- PKB main
- Procedure Index Table
- Variable Index Table
- Statement Type Lists
- Pattern Table
- Follows/Follows* Table

- Parents/Parent* Table
- Uses Table
- Modifies Table
- Calls/Calls* Table
- Next/Cache Table for Next*

PKB-Main: PKB-Main controls and stores all the tables in the entire PKB component, it will be used by only SPA front-end parser and query evaluator components in order to reduce coupling. The APIs of the PKB-Main plays a central role in the SPA architecture.

Procedure Index Table

The Procedure Index Table stores all the procedures and their respective indexes. Before the parser starts parsing the SIMPLE source code, the procedure index is initialized to an arbitrary value of 0. Whenever a new procedure is added to its table by the parser, an index will be generated by the Procedure Index Table and used as a key for the procedure name. Its structure is as follows:

ProcIdxTable.h

- unordered_map<string procName, int procIndex> procIdxMap
- unordered_map<int procIndex, string procName> procNameMap
- int procIdx (initialized to 0, to generate indexes for procedures)

Variable Index Table

The Variable Index Table is stored similarly as the Procedure Index Table. Its structure is as follows:

VarIdxTable.h

- unordered_map<string varName, int varIndex> varIdxMap
- unordered_map<int varIndex, string varName> varNameMap

- `int varIdx` (initialized to 0, to generate indexes for indexes)

Statement Type List

The Statement Type List stores all the statement numbers added by the Parser, according to the respective entities. For example, if the statement number is an assignment statement, it is stored in the **assignList**. Similarly, statement number of the while statement is stored in **whileList**. As such, when QueryEvaluator asks for assign or while statements, we are able to either retrieve the statement numbers that belong to a requested entity by filtering away unwanted data.

Its structure is as follows:

StmtTypeList.h

- `list<int> allStmtList`, to store statement numbers with entity 'STMT'
- `list<int> assignStmtList`, to store statement numbers with entity 'ASSIGN'
- `list<int> whileStmtList`, to store statement numbers with entity 'WHILE'
- `list<int> ifStmtList`, to store statement numbers with entity 'IF'
- `list<int> callsStmtList`, to store statement numbers with entity 'CALLS'

Uses Table

The Uses Table stores all the Uses relationships extracted by the Parser in the form hash maps (`UsesTableStmtToVar`) where the integer statement number or integer procedure index are keys, and a list of integer variable indexes are values. Firstly, when the design abstraction for Uses is added by the parser, the raw variable (or procedure) input in the form of string is converted to a variable (or procedure) index. Both the statement number and resultant integer index is then added to the hash map. Additionally, PKB also adds the reverse relationship internally to another hash map called `UsesTableVar`, where integer variable (or procedure) indexes as stored as a key, and statement numbers as a value. In

the file UsesTableVar, it contains multiple maps such that the design abstraction can be stored according to the entity of the statement, so as to facilitate retrieval when queries such as Uses(a,v) and Uses(w, v) are called by the QueryEvaluator.

UsesTableStmtToVar.h

- unordered_map<int stmtNum, list<int varIdx> > usesStmtToVarMap

UsesTableProcToVar.h

- unordered_map<int procIdx, list<int varIdx> > usesProcToVarMap

UsesTableVar.h

- unordered_map<int varIdx, list<int stmtNum> > usesVarToStmtMap
- unordered_map<int varIdx, list<int assignNum> > usesVarToAssignMap
- unordered_map<int varIdx, list<int procIdx> > usesVarToProcMap
- unordered_map<int varIdx, list<int whileStmtNum> > usesVarToWhileStmtMap
- unordered_map<int varIdx, list<int ifStmtNum> > usesVarToIfMap

Modifies Table

The Modifies Table is populated similar to the Uses Table.

ModTableStmtToVar.h

- unordered_map<int stmtNum, list<int varIdx> > modStmtToVarMap

ModTableProcToVar.h

- unordered_map<int procIdx, list<int varIdx> > modProcToVarMap

ModTableVar.h

- unordered_map<int varIdx, list<int stmtNum> > modVarToStmtMap
- unordered_map<int varIdx, list<int assignNum> > modVarToAssignMap

- `unordered_map<int varIdx, list<int procIdx> > modVarToProcMap`
- `unordered_map<int varIdx, list<int whileStmtNum> > modVarToWhileStmtMap`
- `unordered_map<int varIdx, list<int ifStmtNum> > modVarToIfMap`

Pattern Table

The Pattern Table stores the patterns of all assignment, while, and if statements extracted by the parser. The pattern is stored as follows:

Key - statement number in the form of integer.

Value - a pair containing LHS variable and a postfix expression of the RHS in that statement number.

When the pattern is added by the parser, the LHS variable is stored as a variable index with a type integer, whereas for the RHS expression, the raw input in the form of infix expression is computed internally into postfix by the Pattern Table. The first argument of pattern while and pattern if clauses are defined as control variables. This table will be accessed when pattern clauses are called by the query evaluator. Postfix is used in this table instead of infix in order to facilitate partial matching of the subexpression.

(E.g. Given an infix expression “a+b*(c-d)”, based on the rule of query evaluation, subexpression “c-d” partially matches using postfix, however, “a+b” does not, even though both expressions are the substrings of “a+b*(c-d)”. Hence partial matching is processed as shown: “a+b*(c-d)” is converted to list(“a”, “b”, “c”, “d”, “-”, “*”, “+”) The subexpressions “c-d” and “a+b” are also converted to list(“c”, “d”, “-”) and list(“a”, “b”, “+”) respectively. list(“c”, “d”, “-”) is the sublist of list(“a”, “b”, “c”, “d”, “-”, “*”, “+”), whereas list(“a”, “b”, “+”) Hence it can be deduced that “c-d” is the sub-expression of “a+b*(c-d)”, whereas “a+b” is not)

The steps below show how the partial matching is processed in general.

1. Raw input in the form of infix sub-expressions are passed to the function called `hasPartialMatch(stmt, expression)`

2. Within the function, the infix sub-expression is converted to postfix.
3. The postfix sub-expression performs the partial match by checking with the stored expression (postfix) that is retrieved from the pattern table.
4. If both expressions match, the function returns true.

The structure is as follows:

- `unordered_map<int stmtNum, pair<int LHS, list<string> RHSexpression> > assignPatternTableMap`
- `unordered_map<int stmtNum, pair<int LHS, int controlVar> > whilePatternTableMap`
- `unordered_map<int stmtNum, pair<int LHS, int controlVar> > ifPatternTableMap`

Follows/Follows* table

The Follows and Follows* Tables store all the Follows and Follows* relationships respectively. When the design abstraction Follows are parsed by the Parser, it adds the the statement number as a key, and the statements the current statement follows as a value (in the form of a `pair<followedBy, follows>`). Follows* relationships are then computed by the Design Extractor The table will be accessed when `Follows(s1,s2)` and `Follows*(s1,s2)` are called by the query evaluator. The structure is as follows:

FollowsTable.h

- `unordered_map<int stmtNum, pair<int followedBy, int follows> > followsMap`

FollowsStarBefore.h

- `unordered_map<int stmtNum, list<int> followedBy> followsStarBefore;`

FollowsStarAfter.h

- `unordered_map<int stmtNum, list<int> follows> followsStarAfter;`

Parent/Parent* table

The Parent and Parent* Tables store all the Parent and Parent* relationships respectively. When the design abstraction Parent is added by the Parser, the statement number is added to the hash map as a key, and the list of its children of its respective statements is added as a value. A reverse version of the hash map is implemented, so as to allow computation of the parent statement of the input. Parent* relationship is then computed by the Design Extractor. The table will be accessed when Parent(s1,s2) and Parent*(s1,s2) are called by the query evaluator. The structure is as follows:

ParentToChildTable.h

- unordered_map<int parent, list<int> children> parentToChildMap

ChildToParentTable.h

- unordered_map<int child, int parent> childToParentMap

ParentToChildStarTable.h

- unordered_map<int parent, list<int> children> parentToChildStarMap

ChildToParentStarTable.h

- unordered_map<int child, list<int> parents> childToParentStarMap

Calls/Calls* table

The Calls and Calls* Tables store all the Calls and Calls* relationships respectively. It also stores the mapping of procedures as that is needed to compute transitive closures of Uses and Modifies relationships. Calls* relationship is then computed by the Design Extractor. The structure is as follows:

CallsTable.h

- unordered_map<int, list<int> > callsProcMap
- unordered_map<int, list<int> > callsProcMapReverse

CallsStarTable.h

- `unordered_map<int, list<int> > callsStarProcMap`
- `unordered_map<int, list<int> > callsStarProcMapReverse`

NextTable and Next* Cache

The Next Table stores all the Next relations. It also acts as an adjacency list for the control flow graph which will be used for the computation of Next* relations in the Design Extractor. The Next* Cache stores all the Next* relations that were previously computed by PQL. This is to save time when retrieving identical Next* queries instead of recomputing them. The Next* Cache is only used within NextTable.h.

NextTable.h

- `unordered_map<int, list<int> > nextMap`
- `unordered_map<int, list<int> > nextMapReverse`
- `NextStarCache NextStarCache`

NextStarCache.h

- `unordered_map<int, list<int> > nextStarMap`
- `unordered_map<int, list<int> > nextStarMapReverse`

3.2.3 Design Extractor

The role of the Design Extractor is to compute the complex relations from the tables that Parser has populated. It computes Parent*, Follows* and Calls* relations as well as the interprocedural Uses/Modifies relations. The Parser will call upon the Design Extractor to process these relations once it is done with parsing the SIMPLE source code.

Computation of Parent*/Follows*/Calls* relations

The computation of these complex relations are relatively similar and use the same al-

gorithm. The Design Extractor iterates through the respective base tables (e.g. ParentsTable, FollowsTable) and appends the information found in these tables to the current list. Using the computation of Parent* relations from the Parent relations table given in

ParentToChildTable		ParentToChildStarTable	
Parent	Child	Parent*	Child*
1	2	1	2, 3, 4
2	3	2	3, 4
3	4	3	4

Figure 5: Parent to Child/ChildStar Table

Fig.4 as an example, starting from statement 1, as the relation Parent(1, 2) is true, we then check if statement 2 is a parent statement. In this case, Parents(2, 3) exists, so 3 is appended onto the list of Child* in the ParentToChildStarTable[1]. We then continue to check the child of the current statement (in this case, statement 2) to see if it has a child statement. Since the relation Parent(3, 4) exists, we continue to append 4 onto ParentToChildStarTable[1]. Next, we check if statement 4 is a parent of any statement. Since there does not exist a child with statement 4 as its parent, all the Children* relationship for statement 1 has been computed. We then continue the same steps for all Parents in the ParentToChildTable.

The reverse relations e.g. ChildToParentStarTable given in Fig.5 is also computed in the Design Extractor to facilitate faster result selection.

Similar to Parent and Parent* relations, Follows/Follows* and Calls/Calls* relations can be computed in the same manner.

Computation of Inter-Procedural Uses/Modifies

Computing Uses and Modifies relations that involve procedures requires a traversal of the

ChildToParentTable		ChildToParentStarTable	
Child	Parent	Child	Parent
2	1	2	1
3	2	3	2, 1
4	3	4	3, 2, 1

Figure 6: Child to Parent/ParentStar Table

Calls graph. Our Calls graph is essentially the table of Calls relationship, which acts as an adjacency list. The Design Extractor does a depth first search to reach the leaf nodes and propagate upwards in post-order fashion. The Design Extractor iterates through every Caller procedure and starts a DFS from it. In a Calls graph, the arrow is directed from a Caller to a Callee (i.e. Calls(A, B)).

In this example, we will start the DFS from procedure A. From A, it will traverse down until procedure E and mark it as visited (highlighted in red). As E is a leaf node, it means that it does not call any procedures hence there are no new additional variables appended to its list of variables used. Once visited, it will reverse back to its parent, in this case procedure D. The Design Extractor then checks if all the procedures called by D have been visited. In this case, it is only E that is visited, and hence it appends the variables that E uses onto its list of variables used (in green).

From D, it returns back to its caller C. C checks if all its callees have been visited and if so, appends their used variables onto its list. If its callee's nodes are not visited yet, we run the DFS on the callee. We do so until we return to node A, the node that we started with for this example.

The reason why this algorithm is run on every caller procedure is to ensure complete

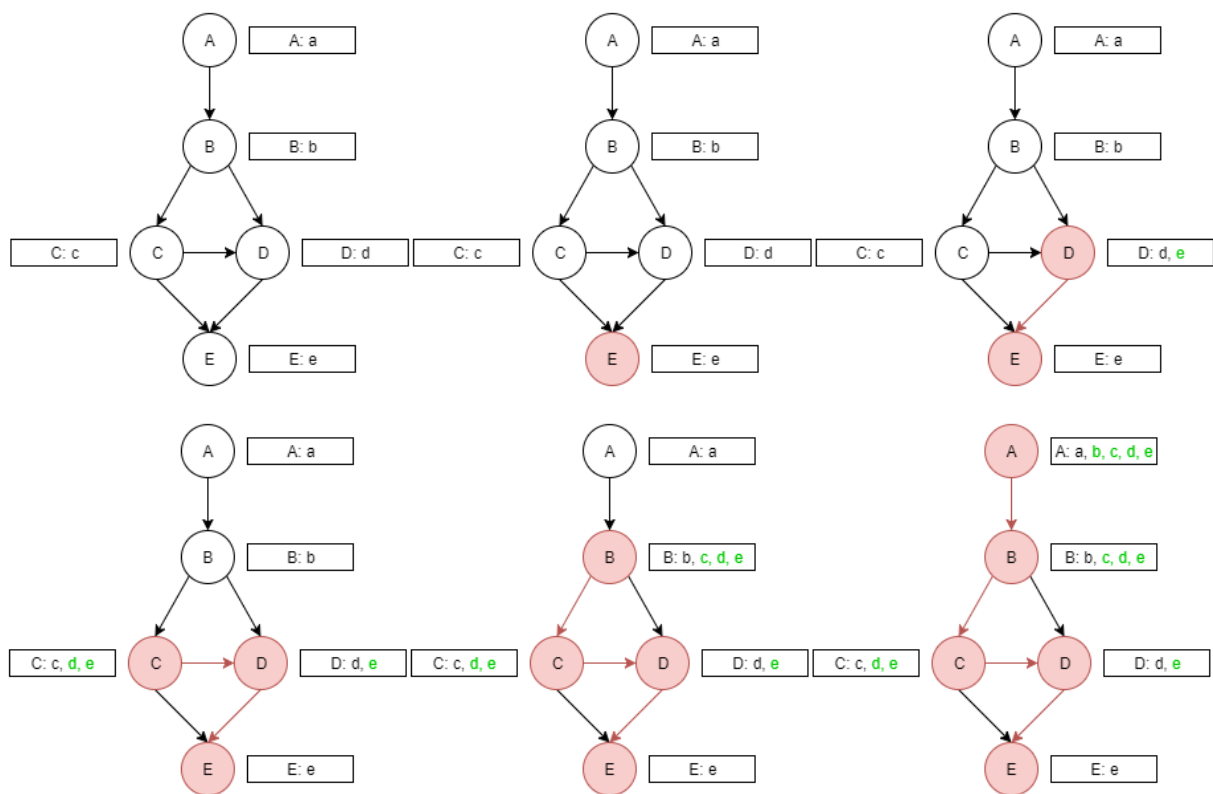


Figure 7: Call Traversal from Proc A

coverage of all Uses in the case where there could be multiple disjoint calls graphs. The same algorithm is run to compute inter-procedural Modifies relations.

Computation of Container Statement Uses/Modifies

After the population of inter-procedural modifies relations, we can then compute Uses and Modifies relations for container statements that contain a call statement in their statement list.

The Design Extractor will retrieve all the Call statements and find the list of Parent* of it. For each Parent*, it will sent that Parent*'s statement to use the variables that are used by the procedure of the Call statement.

Consider the following source code

```
procedure A {  
1   while a {  
2       while c {  
3           call B;  
           }  
       }  
}  
  
procedure B {  
4   x = a + b + c;  
}
```

The Design Extractor will get the list of call statements, in this case only statement number 3. Then, it will get all the Parent* of statement 3 which are 2, 1. For all the Parent* statements, 2, 1, their use relations would be updated with the variables used by statement 3, a, b, c.


Statement	Variables	Statement	Variables	Statement	Variables
1		1		1	a, b, c 
2		2	a, b, c	2	a, b, c
3	a, b, c	3	a, b, c	3	a, b, c
4	a, b, c	4	a, b, c	4	a, b, c

Figure 8: Statement to Variable Tables

The same idea used in computing Uses relation and Modifies relation for container statements.

3.2.4 PQLMain

The PQLMain is the main class that other components interact with. PQLMain employs the **Facade** pattern, whereby other components that need to use PQLMain do not need to know the implementations and the sub-components inside PQLMain. When it is initialised, it takes in a query string and instantiates a blank QueryTree (the main storage component) which is used throughout the process of query processing and evaluation. Fig.7 shows the component diagram of the PQLMain and its sub-components.

3.2.5 Query Validator

The QueryValidator ensures that the query received from PQLMain is syntactically and semantically correct. A series of validation steps are executed while parsing the query to check whether the structure of the query is well-formed and whether the arguments of each individual clause are declared and consist of permitted argument types as described in the program design model. It relies heavily on regular expressions (Regex) that are

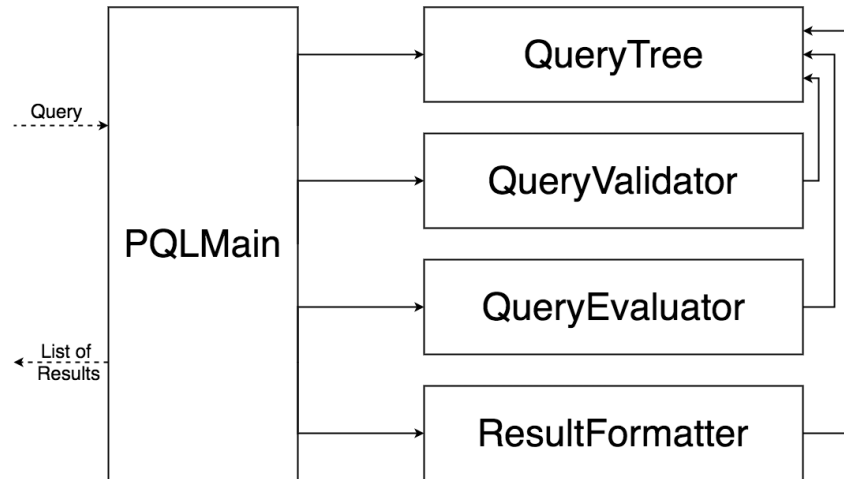


Figure 9: PQLMain Facade Component diagram

aligned to the PQL grammar. As the QueryValidator performs its validation routine, it also helps to build an internal storage structure, the QueryTree, by storing only essential parts of the query. This watered-down version of the query will ease the work of the QueryEvaluator.

The steps parsing the query given below are as follows:

```
stmt s; assign a, a1 , a2;
Select s such that Follows (s, a) pattern a( __, __ ) with a.stmt# = s.stmt#
```

1. Upon receiving a query from PQLMain, the QueryValidator will attempt to dissect the query into Declaration and Selection. This can be done by splitting the query with semi-colons as the delimiter. As each valid query will have its Selection as the last token, any token before the last will be considered as Declaration.

<pre>stmt s assign a, a1 , a2</pre>	}	<i>Declaration</i>
<pre>Select s such that Follows (s, a) pattern a(__, __) with a.stmt# = s.stmt#</pre>	}	<i>Selection</i>

Figure 10: State of query after dissection

2. The QueryValidator will first process each token in Declaration. Each token is further split it into sub-tokens using whitespace as the delimiter. The first sub-token

assign	a,	a1	,	a2
--------	----	----	---	----

will be compared against the acceptable entity as defined in the PQL grammar. If it is invalid, the QueryValidator will immediately return false. If it is valid, it proceeds, and for all subsequent sub-tokens, if any of them are invalid, the QueryValidator will return false. Otherwise, the QueryValidator will count the number of commas in all the declaration statements and compare them with the expected number of commas. One observation made by the team was that the difference between the number of commas and the sum of entity and synonyms is 2. Using this fact, the QueryValidator finishes validation for multiple declarations of the same entity type. For every valid synonym recognized, the QueryValidator inserts it, together with the entity, into the QueryTree. For the above example, the expected number of commas is 2 and the sum of entity and synonyms is 4.

3. After processing Declaration, the QueryValidator will compare Selection against the overall regex definition derived from the PQL grammar. This regex picks up clause group as shown below

Select s	such that Follows(s, a)	pattern a(, _)	with a.stmt# = s.stmt#
----------	-------------------------	----------------	------------------------

4. With the query being partitioned into distinctive clauses, the QueryValidator validates each clause independently. For each clause, the QueryValidator passes it to SelectionValidator. As the query has already passed the overall regex check, there is no need to look for clause keywords to determine which clause they belong to, hence clause keywords ('select', 'such that', 'pattern', 'with' and 'and') can be dropped.



Figure 11: QueryValidator Selection: State of clause before SelectionValidator passes it to the respective handlers

The SelectionValidator will then pass the processed clause to the respective clause handler that is supposed to handle that type of clause. The Handler will then delegate the respective sub-validators to validate and extract the arguments and types of argument. The Handlers adopts the **Strategy** pattern to determine which sub-validator to pass the clause to. Each sub-validator extracts the argument and enquires the QueryTree for the

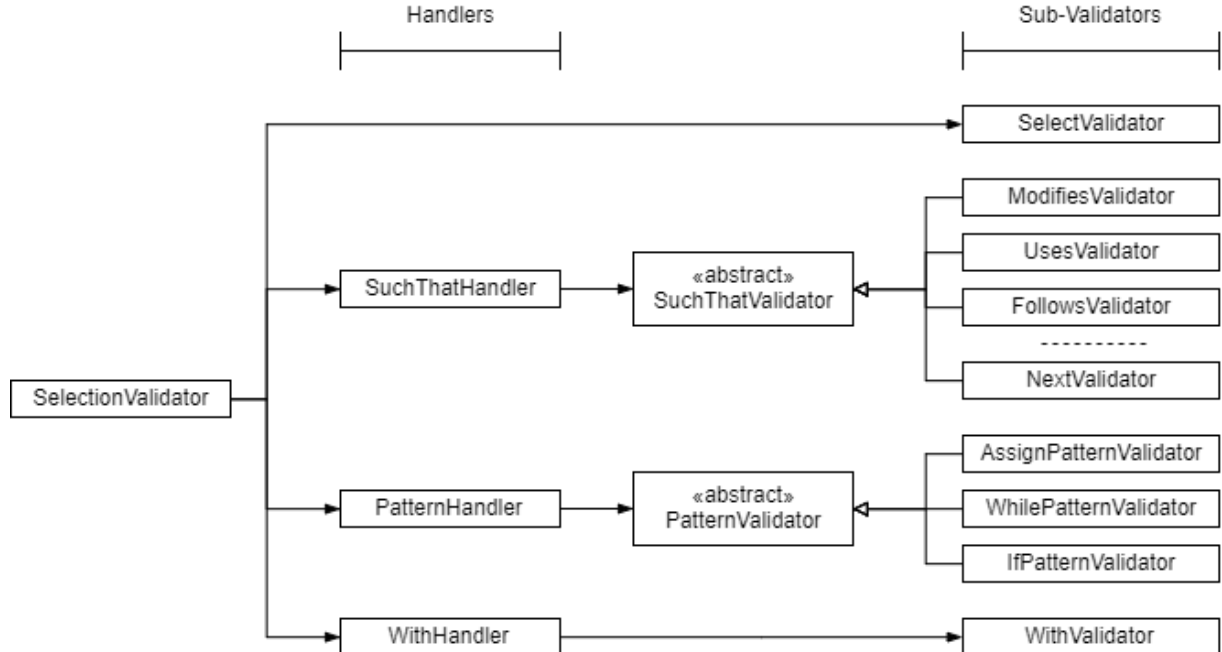


Figure 12: QueryValidator Handler: Possible path taken

type of argument. Once the sub-validator receives the information from the QueryTree, it will tag each argument to its appropriate type.

Arguments are tagged according to their entity type, specific to their respective clauses. The tagging system makes use of enums.

enum PatternType	
ASSIGN_PATTERN	0
WHILE_PATTERN	1
IF_PATTERN	2

Table 1: Relationship Enum

enum PatternType	
ASSIGN_PATTERN	0
WHILE_PATTERN	1
IF_PATTERN	2

Table 2: Pattern Enum

enum Attribute	
STMT_ATTRIBUTE	0
ASSIGN_ATTRIBUTE	1
CALL_STMT_ATTRIBUTE	2
CALL_PROC_ATTRIBUTE	3
WHILE_ATTRIBUTE	4
IF_ATTRIBUTE	5
VARIABLE_ATTRIBUTE	6
CONSTANT_ATTRIBUTE	7
INTEGER_ATTRIBUTE	8
IDENT_WITH_QUOTES_ATTRIBUTE	9
PROG_LINE_ATTRIBUTE	10

Table 3: Attribute Enum

Upon validation and extraction, the sub-validators will return their results to the Handler. The generation of ClauseObject is done via the Factory pattern, using the information retrieved from each sub-validatos. Handler will encapsulate this information into a ClauseObject before storing it in the QueryTree.

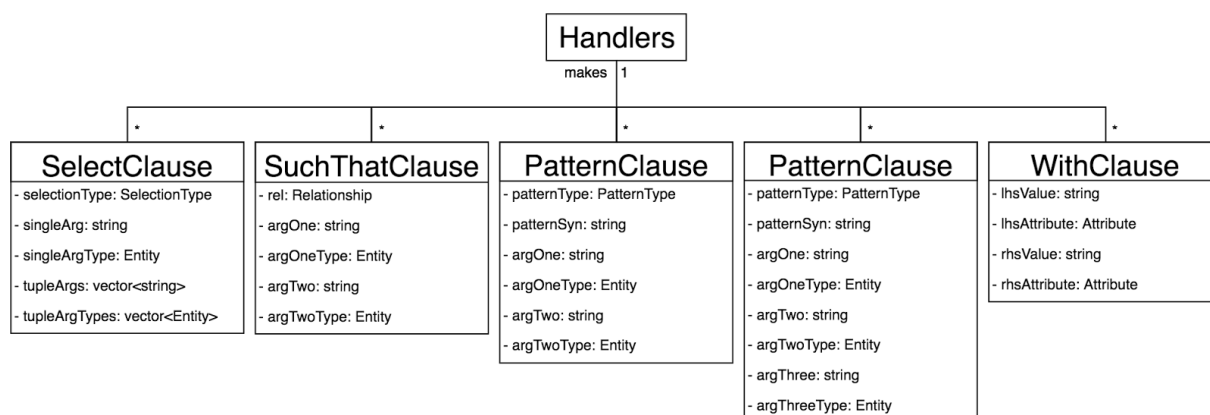


Figure 13: Query Validation Handler

The decision making at each stage is shown in Fig. 12 above.

3.2.6 Query Optimizer (plan)

The QueryOptimizer will be designed to group the different clauses to get an optimal evaluation time in the QueryEvaluator. The QueryOptimizer will work on the query after finishing query validation and before starting evaluation. The main heuristic for optimization is to avoid cross-products, which is a time consuming operation. Thus, the team's tentative plan is to first group the different clauses according to the following rules:-

Group 0: Clauses that return boolean results (e.g Follows(1,2), Uses(1,"x"))

Group 1: Clauses containing synonyms that are not required by the Select clause (e.g Select s such that Follows(s,s1) and Follows(s3,s4) will put Follows(s3,s4) in Group 1)

Group 2: Clauses containing synonyms required by the Select clause for intermediate results.

For 'with' clauses, the QueryOptimizer will implement overwriting of clauses (e.g if the query is procedure p; Select s such that Modifies(p,"x") with p.procName = "Pikachu", the such that clause that will be passed to the evaluator will be Modifies("Pikachu","x").

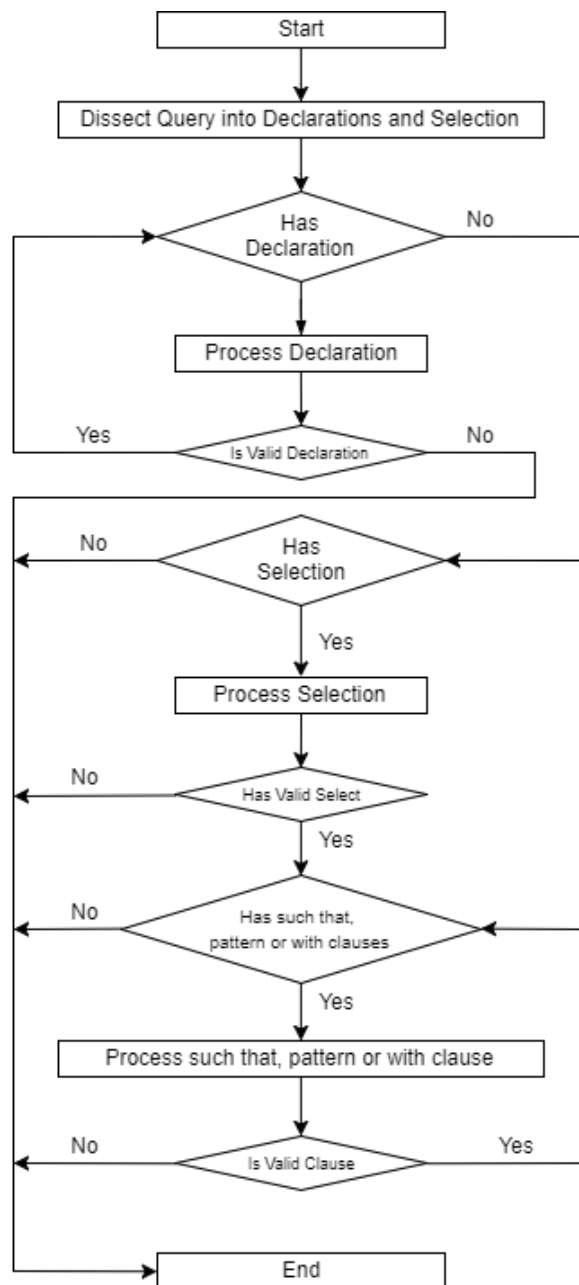


Figure 14: Validator Flowchart

3.2.7 Query Evaluator

The QueryEvaluator is used to retrieve design entities from the PKB that fulfill query specifications. It first retrieves the Clause objects (created by the QueryValidator) from the QueryTree. For each Clause object, the QE will evaluate the clause with the help of appropriate ClauseEvaluators and proceed with the evaluation. These evaluators assist the QE in retrieving the relevant results from the PKB and are produced by providing Clause object to the ResultsFactory. We first discuss the intermediate data structures that we utilise, before providing an in-depth discussion of the evaluation steps taken by the QE.

ClauseResult

The ClauseResult is responsible for storing and maintaining the intermediate results

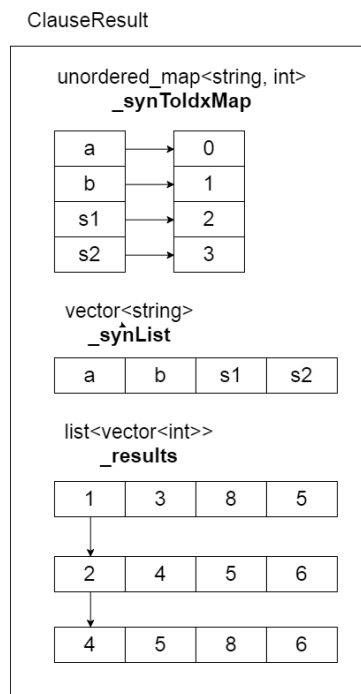


Figure 15: ClauseResult

across Clause objects as they are evaluated by the QueryEvaluator. The ClauseResult maintains several data structures to achieve this purpose.

The intermediate results are stored in a `list<vector<int> >` data structure, where each `vector<int>` represents one combination of all synonyms that satisfies the clauses evaluated so far. A vector is used to store these combinations because it allows access of the values of selected synonyms in $O(1)$ time with indexing. This means that every index position in the vector is mapped to a certain synonym, and this mapping is the same for all vectors in the list. A list is used to store these vectors because combinations of synonym values will need to be inserted and removed from the middle of the list often, and a linked-list can carry out these operations in $O(1)$.

As the index positions in the aforementioned vectors need to be mapped to synonyms, an `unordered_map<string, int>` is used to store the mapping between synonyms and their indices, and a `list<string>` is used to implicitly store the reverse mapping.

Evaluation Process

The evaluation steps are listed below:

As per the **Strategy** pattern, each clause in the clause group will have an evaluator created specially for its evaluation. Figures 16,17 and 18 show an overview of the various evaluators, derived from the base class `ResultEvaluator`.

1. When invoked, the evaluate method of the QE retrieves the vector of Clause objects from the QueryTree.
2. The QE then initialises the ResultFactory and feeds the Clause objects, one at a time, to the ResultFactory to be processed. The Clause objects will be processed in the order of SuchThatClause, PatternClause, followed by WithClause and lastly SelectClause. At any point, if one clause should return false due to invalid evaluation or all results get removed from the ClauseResult, the whole evaluation process will be discontinued.
3. As per Strategy pattern, the ResultFactory will create evaluators specially for the

evaluation of the clauses. Figures 16,17 and 18 show the overview of the various evaluators that can be created.

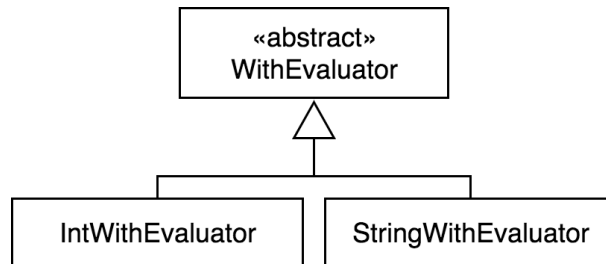


Figure 16: With Evaluator

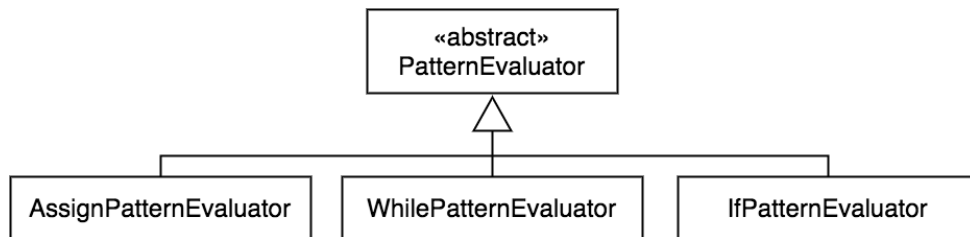


Figure 17: Pattern Evaluator

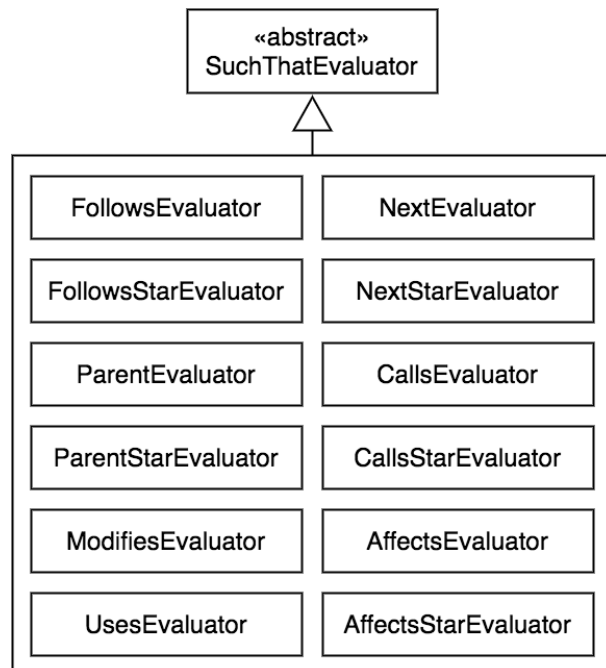


Figure 18: Such That Evaluator

Generation of an evaluator is done via the Factory pattern, using the information retrieved from Clause objects. The appropriate evaluator is chosen based on the type reference of the Clause objects. For example, the SuchThatEvaluator has a relationship type specifier while the PatternEvaluator has a pattern type specifier.

4. Evaluators determine how to evaluate results by passing the arguments of a clause through several cases. Each clause will have its own cases to evaluate depending on the type of inputs that they allow. Table 5 shows an example of the evaluation cases for the Follows relationship.

	Argument One	Argument Two	Result to be merged
Case 1	INTEGER	INTEGER	BOOLEAN
Case 2	INTEGER	UNDERSCORE	BOOLEAN
Case 3	INTEGER	SYNONYM	BOOLEAN, STMT#
Case 4	UNDERSCORE	INTEGER	BOOLEAN
Case 5	UNDERSCORE	UNDERSCORE	BOOLEAN
Case 6	UNDERSCORE	SYNONYM	BOOLEAN, STMT#
Case 7	SYNONYM	INTEGER	BOOLEAN, STMT#
Case 8	SYNONYM	UNDERSCORE	BOOLEAN, STMT#
Case 9	SYNONYM	SYNONYM	BOOLEAN, STMT#, STMT#

Table 4: Follows Evaluator Cases

5. After the specific evaluators evaluate the clauses, the results of the current clause will be merged into the ClauseResult object, which keeps track of the intermediate results. For purely Boolean cases (i.e. Follows(3, 4); Case 1 in Table 5), only the validity is returned by the evaluator, and this determines whether evaluation will proceed with the next clause. The validity replaces any previously stored boolean value. Recall that the evaluating process will be terminated if the stored validity is false; thus a false stored validity will never be overwritten. For cases with Synonyms

, the validity is returned, and results are stored or modified in the ClauseResult via its reference pointer.

6. For clauses that contains synonyms, there are 5 possible cases that may arise.

Cases	Description
1 new synonym	There is 1 synonym in the clause which is new to the ClauseResult
1 existing synonym	There is 1 synonym in the clause that already exist in the ClauseResult
2 new synonyms	There are 2 synonyms in the clause, both are new to the ClauseResult
2 existing synonyms	There are 2 synonyms in the clause, both exist in the ClauseResult
1 new synonym and 1 existing synonym	There are 2 synonyms in the clause, one is new to the ClauseResult and one exist in the ClauseResult

Table 5: Cases of Clauses with Synonyms

For the cases of 1 new synonym and 2 new synonyms, the way the ClauseResult works depends on whether there are already other synonyms existing in the ClauseResult. These two scenarios are illustrated in the following diagram.

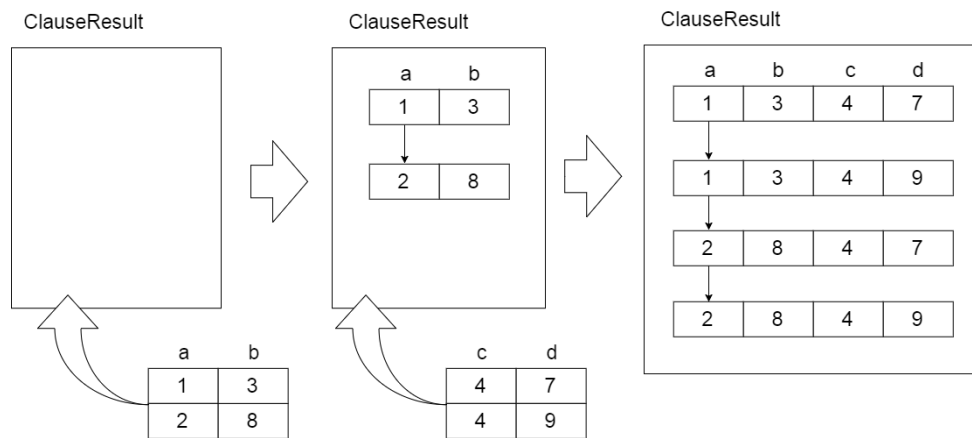


Figure 19: Adding New Single Synonym Results

Notice that for the case of 2 new synonyms, PKB will need to compute all pairs of the two synonyms for evaluator, which is an expensive process. Optimisation to avoid this scenario is possible, which will be introduced in future iterations.

For the cases of 1 existing synonym and 2 existing synonyms, the ClauseResult will loop through the existing results and remove the ones that do not have a match in the new result list. For the case of 1 new synonym and 1 existing synonym, the

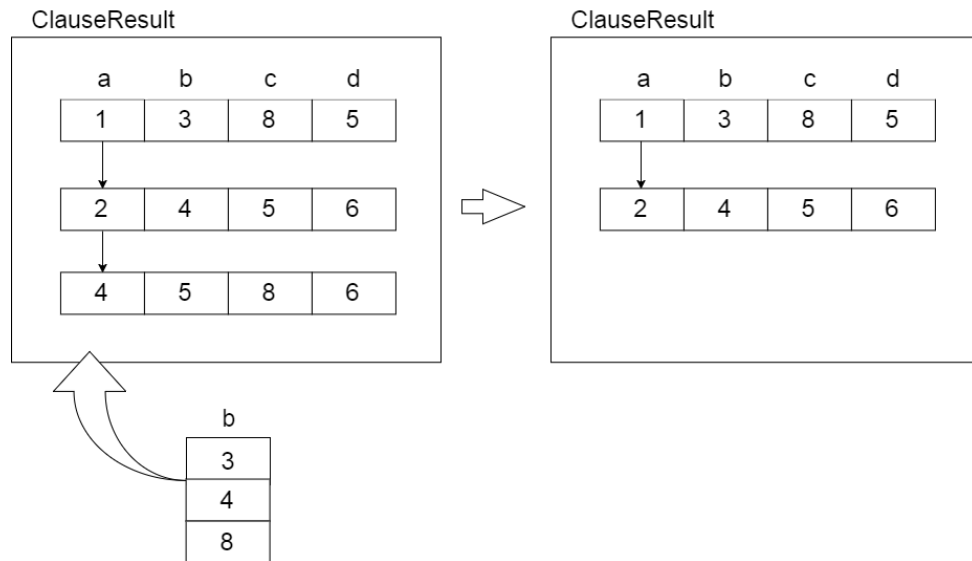


Figure 20: Adding Existing Synonym Results

ClauseResult will create an `unordered_map<int, list<int>>` to help with the merg-

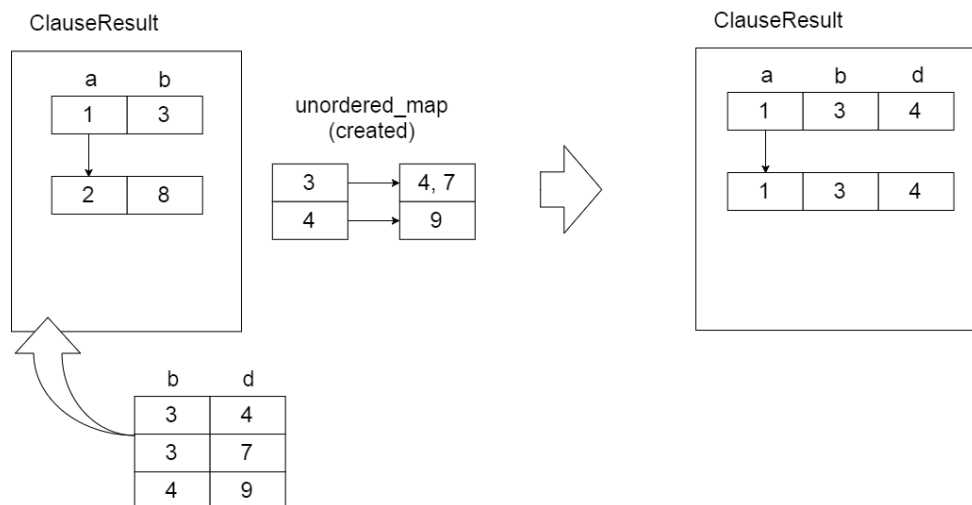


Figure 21: Adding A New Synonym Paired to an Existing Synonym

ing process, this `unordered_map` will map each values of the existing synonyms to the list of the corresponding values of the corresponding new synonym. For each of

the existing result combination, if the common synonym's value has a match in the `unordered_map`, the new synonym's value will be appended to the combination. If the common synonym's value has no match in the `unordered_map`, the combination will be discarded.

If the number of combinations of the existing results is M , and that of the results to merge is N , the time complexity of this merging process would be $O(M+N)$, which is sufficiently fast.

7. After the `ResultFactory` has finished processing all the `Clause` objects and all of them returns true for validity, the factory will call the `makeClauseResult()` method to return the final `ClauseResult` object back to `QE`. `QE` will then store this back into the `QueryTree`, waiting for it to be picked up by the `ResultFormatter` to format and return the final result.

3.2.8 Result Formatter

The `ResultFormatter` uses the `QueryTree` and the `ClauseResult` to return results in the form of a list of strings (`list<string>`) back to the UI. It first checks the type of the `Select` clause (whether it's `BOOLEAN`, single synonym or tuple).

1. If the query selects `BOOLEAN`, the `ResultFormatter` checks whether the `ClauseResult` has results and if `QueryTree` has any clauses to evaluate. If `QueryTree` contains clauses to evaluate, if `ClauseResult` doesn't have results after evaluation, then the `ResultFormatter` returns a 'false' result; it returns 'true' for all other cases.
2. If the query selects a single synonym/tuple, then `ResultFormatter` uses `ClauseResult`'s API `getSynonymResults()` to get the raw results. If the synonym being selected returns a numerical result, then the list of integers (`list<int>`) is converted to a list of strings (`list<string>`) and projected to the UI. In case the synonym being

selected needs to return a string result (e.g entities of the type VARIABLE and PROCEDURE) the list of integers received from ClauseResult are converted to a list of strings according to the mapping done by PKB (as previously, we map all strings to ints to optimize Query Evaluator; thus, we need to convert the mapped ints to strings when we output the results)

3.3 Design decisions

3.3.1 Parsing Strategy

Problem	<p>The Parser has a lot to do while parsing the SIMPLE source code:</p> <ul style="list-style-type: none">- Various types of syntax validation- Display error message as user feedback when syntax error is detected- Extract various design abstractions and populate PKB
Alternatives	<ol style="list-style-type: none">1. Do all of the job of the parser in one parse2. Do many parses, each parse doing one particular task, e.g. bracket-matching, Calls relation, Uses relation, etc.3. Hybrid of the above two alternatives.
Criteria for comparison	<ol style="list-style-type: none">1. Time-efficiency2. Ease of implementation3. Testability4. Maintainability5. Extensibility to iteration 2, iteration 3, and possible bonus features
Decision	<p>Choice: Hybrid</p> <p>Doing everything in one parse can be difficult to implement and error-prone. It also makes testing difficult. On the other extreme, having many parses, each doing one simple task will also result in a lot of duplicate codes, which will make parser difficult to be maintained and extended because if there is a need to change something, all the duplicated code will need to be changed.</p> <p>Therefore, using just a few parses with each parse doing related tasks is probably the most reasonable design. The Parser will do one parse just to check whether braces are paired up correctly, and do a second parse to extract design abstractions and populate PKB.</p>

3.3.2 PKB Design: Representation of Design Abstraction

Problem	Speed is a very important aspect for query evaluation. If this aspect is neglected, it would cause a longer runtime when multiple queries are processed. Therefore, design of the data structures to store and represent design abstractions added by the parser is extremely crucial to ensure that data is accessible at the shortest time possible.
Alternatives	<ul style="list-style-type: none">- AST- Tables in the form of hash maps
Criteria for comparison	<ol style="list-style-type: none">1. Time complexity of retrieval, addition and deletion2. Simplicity of implementation3. Extensibility to future iterations and possible bonus features4. Testability
Decision	<p>Choice: Tables in the form of hash maps</p> <p>AST is the conventional ways to store parsed data into. However, when it comes to accessing the data by the Query Evaluator component, it has to perform traversing of the tree, which could take a lot of time during query processing. Also, doing modifications to allow more design entities would be more complicated. Alternatively, hash maps can be used to store data. Having multiple hashmaps rather than AST allows faster searching, addition, deletion of data.</p>

The table below shows the time complexity of query processing when AST and tables are

used. 's' represents the number of statements in the SIMPLE programme):

	Using AST	Using Tables
Data searching	$O(\log s)$	$O(1)$
Retrieving single data	$O(\log s)$	$O(1)$
Retrieving all the data	$O(s \log s)$	$O(s)$

3.3.3 PKB Design: Evaluating reverse relationships

Problem	For queries that require retrieval of the left hand parameter (as in x in $Uses(x,y)$, $Modifies(x,y)$, etc.), the key of the hash map needs to be retrieved from the input value. For example, if the query evaluator calls $Follows(3, s)$, statement s can be retrieved in $O(1)$ time, by using the key '3'. However, if $Follows(s, 3)$ is called, since 's' is an unknown key, it is not possible to retrieve in constant time. Therefore, traversing is needed to find the key that matches with the value, and it could take $O(n)$ time complexity at its worst.
Alternatives	<ul style="list-style-type: none">- Add another hash maps to store reverse relationships- Self-balancing binary search tree (BST)
Criteria for comparison	<ol style="list-style-type: none">1. Time complexity for retrieval2. Ease of implementation
Decision	<p>Choice: Add another data structure to store reverse relationships</p> <p>The time complexity of searching in hash maps is faster compared to searching in trees, as traversing of trees takes $O(\log n)$ time. A drawback in this method is that adding an extra data structure doubles the space complexity, but the priority of this SPA is the speed and efficiency of data retrieval, rather than the size of the program. Therefore, sacrificing memory space for better speed is more practical.</p>

3.3.4 PKB Design: CFG Design

Problem	Storage of CFG
Alternatives	<ul style="list-style-type: none">- Adjacency List- Adjacency Matrix- Edge List
Criteria for comparison	<ol style="list-style-type: none">1. Time complexity for finding relation2. Time complexity for traversal3. Ease of implementation
Decision	<p>Choice: Adjacency List</p> <p>Time complexity for finding relation: Edge List $O(E) >$ Adjacency List $O(2) = O(1) =$ Adjacency Matrix $O(1)$.</p> <p>Time complexity for traversal: Edge List $O(E) >$ Adjacency Matrix $O(V^2) >$ Adjacency List $O(V)$</p> <p>Time complexity for finding relation in Adjacency List in our CFG is $O(1)$ as each statement in the CFG can have at most 2 vertices. Therefore, Adjacency List is the best for retrieval of a relation as well as for graph traversal. At the same time, the way PKB stores next relations are already in an Adjacency List format. Hence it is easier to use the table of all next statements as our Adjacency List.</p>

3.3.5 Pattern Matching: Representation of expressions

Problem	Pattern matching requires consideration of the precedence of operators and associativity, therefore it cannot be performed by matching substrings. Therefore, an alternate representation of expressions is required to facilitate pattern matching.
Alternatives	<ul style="list-style-type: none">- Trees to represent expression- Postfix matching
Criteria for comparison	<ol style="list-style-type: none">1. Time complexity of retrieval2. Simplicity of implementation3. Testability
Decision	<p>Choice: Postfix matching</p> <p>Postfix matching is used instead of trees due to the fact that trees are harder to implement and make debugging more time consuming. Postfix expressions can be represented in a list of strings containing operator/operands, and they can be easily inserted into and retrieved from hash maps. Pattern matching can be done by comparing sublists.</p>

3.3.6 Query Validation Design

Problem	<ol style="list-style-type: none">1. Difficulty in parsing query2. Too many if-else string comparisons
Alternatives	<ol style="list-style-type: none">1. Usage of Regular Expression2. Naive tokenizing with specific delimiter3. Naive if-else
Criteria for comparison	<ol style="list-style-type: none">1. Time-efficiency2. Ease of implementation3. Testability4. Maintainability5. Extensibility to iteration 2, iteration 3, and possible bonus features
Decision	<p>Choice: Regular Expression</p> <p>Parsing the query may be tedious as there are many considerations to take into account to when choosing the delimiters. Regex provides a fast and elegant solution. One line of regex can replace a hundred lines of procedural code. As members of the group are well-versed and experienced with regex, implementation was fast. Regex is easier to maintain, especially when they are stored in string constants that can be reused and concatenated to form a larger expression.</p>

Problem	Many “if-else” statements to handle the validation of clauses
Alternatives	Strategy pattern (Handlers and sub-validators)
Criteria for comparison	<p>Pros</p> <ol style="list-style-type: none">1. Easy modification and extension as understanding the individual strategy to process each clause does not require you to understand the entire process.2. Less tedious to maintain and extend the code as changes only need to be done in a class design as compared to all classes.3. Eliminates conditional statements for selecting the correct validator to process each clause. <p>Cons</p> <ol style="list-style-type: none">1. Increase overhead in terms of memory space and execution time.2. Increase number of classes to handle the validators.3. Increase complexity of the solution.

3.3.7 Query Evaluation Design

Problem	Many “if-else” statements to handle the evaluation cases of all the clauses, deeply nested nested if else blocks makes extension of the evaluators very tedious
Alternatives	<ol style="list-style-type: none">1. Strategy pattern (ClauseEvaluator and sub-classes)2. Factory pattern (ResultFactory and ClauseResult)
Criteria for comparison	<p>Pros: Strategy Pattern</p> <ol style="list-style-type: none">1. Modifying or understanding the individual strategy to process each clause does not require you to understand the entire process.2. Less tedious to maintain and extend the code as changes only need to be done in a class design as compared to all classes.3. Eliminates many conditional statements for evaluator to select the correct function to process each clause. <p>Pros: Factory Pattern</p> <ol style="list-style-type: none">1. Lazy initialization to delay creation of the subunit processors to process each clause.2. A single point of control to create the subunit processor, which are evaluators. <p>Cons</p> <ol style="list-style-type: none">1. Increase overhead in terms of memory space and execution time.2. Increase number of classes to handle the validators.3. Increase complexity of the solution.

3.3.8 Data Structures for Intermediate Result

Problem	<p>The data structures of intermediate result is crucial for the good performance of PQL queries as numerous operations will need to be carried out on the intermediate result when evaluating each clauses. These operations include merging of tables, insertions, deletions, and many more, which might be computationally expensive if not optimised.</p>
Alternatives	<p>Representation of results:</p> <ol style="list-style-type: none">1. Actual form (e.g. integer for statement, string for variable/procedure names)2. Indices
Criteria for comparison	<ol style="list-style-type: none">1. Time-efficiency2. Ease of implementation3. Testability4. Maintainability5. Extensibility to iteration 2, iteration 3, and possible bonus features
Decision	<p>Choice of representation of results: Indices</p> <p>Matching indices is a lot faster than string-matching.</p> <p>Choice of data structures: Linked-list of vectors</p> <p>Using a linked-list will allow very fast insertion and deletion of existing combination of synonym results in the middle of the list. This will enhance performance tremendously compared to using a vector of vectors.</p>

4 Documentation and Coding standards

We follow a combination of Coding Standards from different sources in order to get an optimal set of rules that help us the most in our project.

4.1 Abstract API and Concrete API

We made our Abstract API as similar as we could to concrete APIs in C++ to avoid confusion. At the same time, we tried to keep our Abstract API programming language independent and made it general enough to be implementable. Ex. Abstract API LIST-STMT was implemented with a concrete API of `list<string>`.

4.2 Naming Conventions

We follow the naming conventions detailed in this website **Geosoft C++ naming conventions**

- Method names should clearly describe their intended functions.
- Camel cases are used for naming of methods, parameters and variable. (E.g `addWhileStmt(int stmt, string controlVar), varIdx`, etc)
- Pascal cases are used for naming of classes and files. (E.g `FollowsTable`, `UsesTableStmtToVar`)
- Prefixes that start with ‘`_`’ are named in private methods. (E.g `_pkbMainPtr`, `_isValidSyntax`)
- Uppercase is used for naming of constants, separated by ‘`_`’ for each word. For example: `REGEX_VALID_ENTITY_NAME`, `INT_INITIAL_PROC_INDEX`

- To represent star relationships, ‘Star’ is used in naming of classes, methods and files. For example Follows* is represented as FollowsStar.

4.2.1 Formatting

We follow the formatting detailed in this website **Google style C++ formatting**

Some standard rules we follow are given below.

- We use 4 whitespaces as our default indentation.
- Every class must contain one .h file and one .cpp file, and the class, .h files and .cpp files must have the same name.
- For use for braces in methods or container statements, we have opening braces and closing braces both on a new line.

4.3 Class hierarchy

We store our SPA components in separate directories, divided into Parser, PKB and PQL. Our design abstractions are stored in separate source files (.cpp) and our concrete APIs are defined in the corresponding header files (.h).

5 Testing

Our testing objectives are:

1. To discover possible defects and loopholes which may arise while developing the SPA
2. To gain confidence in developing and maintaining the SPA
3. To ensure the final product meets the SPA requirements

5.1 Test Plan

The testing process consists of 3 stages, namely Unit Testing, Integration Testing and System Testing. For every new functionality added or modification made to the SPA, these tests are rerun. This ensures that newly added function produces the expected result. Regression testing is conducted to ensure that the changes do not affect previously tested functionalities. With all these tests put in place, the code remains buildable at all times. If there were any bugs, it can be resolved promptly.

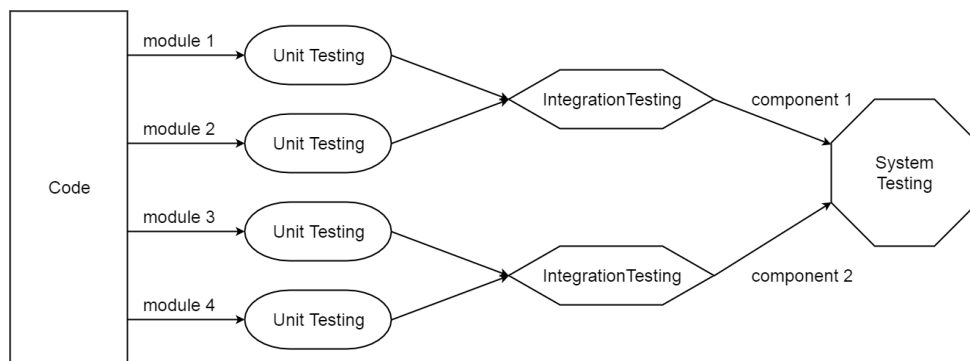
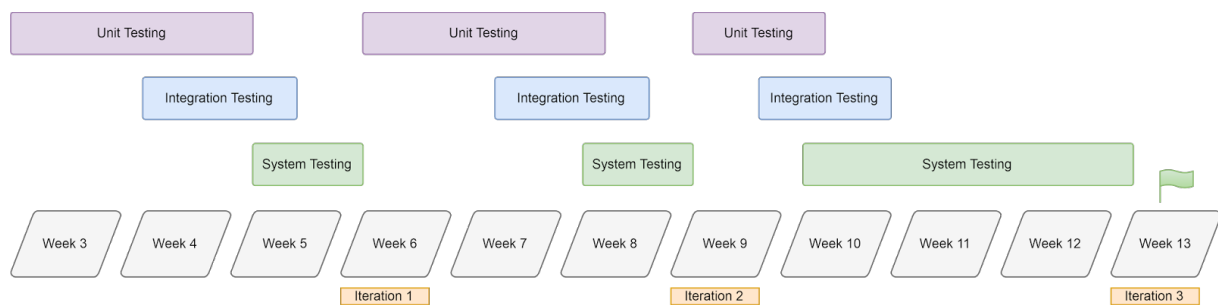


Figure 22: Flow chart showing the testing process

5.1.1 Test Schedule for All Iterations



For each iteration, unit testing will be conducted during the first half of the iteration, to ensure that the new modules introduced produced the expected result. Algorithms are verified and logical errors will be corrected during this process. Private methods are tested by creating friend classes and accessing them using these friend classes. Individual

developers are responsible for testing their own code. Integrating testing will be conducted to ensure that these modules are able to communicate and interact with other components as expected. For example, tests are written to check whether QueryValidator is able to store data into the QueryTree, and whether the retrieved data from QueryTree are the same as those stored by the QueryValidator. Towards the end of each iteration, system testings are carried out frequently using the AutoTester to check for regression. Two developers are delegated to creating new source programs and generating queries that covers corner cases. At any point during the testing process, bugs that surface will be resolved immediately by the developer of the component.

5.1.2 Test Objectives

The following components are the components subjected to testing:

5.2 Test Utilities

5.2.1 Test Utilities Class

Test utilities classes provide helper methods to aid the testing process.

5.2.2 Test Objectives

The following components are subjected to testing:

5.2.3 Script for automating System Testing

Scripts are created to automate system testing.

<u>Component</u>	<u>Objectives</u>
Parser	<ul style="list-style-type: none">- To ensure that the source file is parsed and validated accurately- To ensure that the correct information is stored into PKB
Design Extractor	<ul style="list-style-type: none">- To ensure that the computation of transitive closure and computation of star relationships are done correctly
PKB	<ul style="list-style-type: none">- To ensure that information can be stored successfully- To ensure that the information retrieved from PKB is accurate and expected
Query Validator	<ul style="list-style-type: none">- To ensure that the query is parsed and validated accurately- To ensure that the query is partitioned accurately- To ensure that the partitions are stored in the <i>QueryTree</i> accurately
Query Evaluator	<ul style="list-style-type: none">- To ensure that the information retrieved from the <i>QueryTree</i> is accurate- To ensure that the result retrieved from the PKB is accurate- To ensure that merged result of various clauses produce the expected result
Query Tree	<ul style="list-style-type: none">- To ensure that the information is stored successfully- To ensure that the information is retrieved correctly

<u>Class</u>	<u>Description</u>
ParserChild	A child class of <i>Parser</i> . It access the protected methods and variables and made them available for unit testing.
UtilityQueryTree	Retrieves information from <i>QueryTree</i> and compare with the data expected to be retrieved. It is mainly used in integration testing of <i>DeclarationValidator</i> and <i>QueryTree</i> .
UtilitySelection	Makes the different <i>clause</i> objects that contain expected data. It is mainly used in the integration testing of <i>QueryValidator</i> and <i>ResultFormatter</i>
FriendDeclarationValidator	Friend class of <i>DeclarationValidator</i> . It access the private methods and variables and made them available for integration testing.

1. Given a SIMPLE source program, a unique excel sheet containing the different queries designed for the source program will be created.

	A	B	C	D	
1	Index	Declaration	Select	Expected Answer	Comment
2	1	stmt s;	Select s	1,2,3,4,5,6,7,8	All statements HaveResult
3	2	assign a;	Select a	1,2,4,5,6,7,8	All assignments HaveResult
4	3	while w;	Select w	3	All whiles HaveResult
5	4	variable v;	Select v	a,b,c,d,i,j	All variables HaveResult

Figure 23: Excel sheet containing the queries

2. A python script will convert this excel sheet into a text file so that it can be passed to AutoTester.
3. A batch file is also written to automatically run the AutoTester.

```
Welcome to Team 11 AutoTester
=====
[0] Generate all queries from xls to txt
[1] Run AutoTester for test 1
[x] Exit
Please select your options:
```

Figure 24: Batch file used for automating AutoTester

5.3 Unit Tests

Unit testing involves the testing of individual modules within the component, as well as the component itself, isolated from other components of the system. It is a form of white-box testing, where test cases specifically targets the internal structure of the module.

5.3.1 PKB Unit Test

Before the start of the test, pre-defined relationships are injected directly into the PKB table.

<u>Test Purpose</u>	To test if PKB computes design entity, stmt, correctly
<u>Required Test Input</u>	<pre>TEST_METHOD (TestGetStmt) { FollowsTable table; Assert::IsTrue(table.addFollows(2, 1, 3)); Assert::IsTrue(table.addFollows(3, 2, 4)); Assert::AreEqual(table.getStmtAft(2), 3); Assert::AreEqual(table.getStmtBef(2), 1); Assert::AreEqual(table.getStmtAft(3), 4); Assert::AreEqual(table.getStmtBef(3), 2); }</pre>
<u>Expected Test Results</u>	true, true, true, true, true, true

5.3.2 PQL Unit Test

In PQL Unit testing, the tests are mostly written to test QueryValidator. In particular, the tests targets the accuracy of the regex of each clause. As QueryValidator determines

<u>Test Purpose</u>	To test if PKB computes relations, CallsStar, correctly
<u>Required Test Input</u>	TEST_METHOD(TestCallsTable) <pre> { PKBMain PKB; PKB.addVariable("a"); PKB.addVariable("b"); PKB.addVariable("c"); PKB.addVariable("d"); PKB.addVariable("e"); PKB.addVariable("f"); PKB.addVariable("g"); PKB.addVariable("h"); PKB.addVariable("i"); PKB.addProcedure("One"); PKB.setCallsRel(2, "One", "Two"); PKB.setCallsRel(3, "One", "Three"); PKB.setUseTableProcToVar("One", "a"); PKB.addProcedure("Three"); PKB.setCallsRel(6, "Three", "Four"); PKB.setCallsRel(7, "Three", "Five"); PKB.setCallsRel(8, "Three", "Six"); PKB.addProcedure("Four"); PKB.addProcedure("Five"); PKB.setCallsRel(14, "Five", "Six"); PKB.setCallsRel(15, "Five", "Seven"); PKB.setUseTableProcToVar("Five", "b"); PKB.addProcedure("Six"); PKB.setCallsRel(19, "Six", "Seven"); PKB.addProcedure("Two"); PKB.setCallsRel(24, "Two", "Eight"); PKB.addProcedure("Seven"); PKB.setCallsRel(28, "Seven", "Nine"); PKB.addProcedure("Eight"); PKB.setCallsRel(33, "Eight", "Nine"); PKB.addProcedure("Nine"); PKB.setUseTableProcToVar("Nine", "c"); PKB.setModTableProcToVar("One", "a"); PKB.setModTableProcToVar("Two", "b"); PKB.setModTableProcToVar("Three", "c"); PKB.setModTableProcToVar("Four", "d"); PKB.setModTableProcToVar("Five", "e"); PKB.setModTableProcToVar("Six", "f"); PKB.setModTableProcToVar("Seven", "g"); PKB.setModTableProcToVar("Eight", "h"); </pre>

	<pre>PKB.setModTableProcToVar("Nine", "i"); PKB.startProcessComplexRelations(); list<int> expectedList = { 5, 4, 2, 0 }; expectedList.sort(); list<int> resultList = PKB.getCallerStar("Seven"); resultList.sort(); Assert::IsTrue(resultList == expectedList); expectedList = { 3, 4, 5, 6, 8 }; expectedList.sort(); resultList = PKB.getCalleeStar("Three"); resultList.sort(); Assert::IsTrue(resultList == expectedList); pair<list<int>, list<int>> allCallsStar = PKB.getAllCallsStar(); Assert::IsTrue(PKB.isCallsStar(0, 2)); Assert::IsTrue(PKB.isCallsStar(0, 3)); Assert::IsTrue(PKB.isCallsStar(0, 4)); Assert::IsTrue(PKB.isCallsStar(0, 5)); Assert::IsTrue(PKB.isCallsStar(0, 6)); Assert::IsTrue(PKB.isCallsStar(0, 7)); Assert::IsTrue(PKB.isCallsStar(0, 8)); Assert::IsTrue(PKB.isCallsStar(1, 7)); Assert::IsFalse(PKB.isCallsStar(1, 6)); Assert::IsFalse(PKB.isCallsStar(1, 4)); Assert::IsTrue(PKB.isCallsStar(1, 8)); }</pre>
<u>Expected Test Results</u>	true, true, true, true, true, true, true, true, false, false, true

the argument type by enquiring the QueryTree, semantic checks for the types of arguments are done in the integration tests.

Test Purpose	To test for invalid number of arguments for Follows relationship
Required Test Input	<pre>TEST_METHOD(TestRegex_Follows_ArgCount_invalid) { string str = "Follows(validArgsSyntax, with, extraArgs)"; Assert::IsFalse(RegexValidators::IsValidFollowsRegex(str)); str = "Follows(insufficientArgs)"; Assert::IsFalse(RegexValidators::IsValidFollowsRegex(str)); }</pre>
Expected Test Results	false, false

Table 6: Unit Test for PQL

Another example will be testing of ClauseResult used by the QueryEvaluator.

Test Purpose	<p>To test the method of ClauseResult (intermediate result) addNewSynPairResults that merges two result tables, namely the existing result and new result, for the case when the clause currently being evaluated is introducing two new synonyms.</p> <p>Cases to consider:</p> <ol style="list-style-type: none">1. Adding new results when ClauseResult is still empty2. Adding new results that is a subset of the results that are already in ClauseResult.3. Adding new results that partially overlaps with the results that are already in ClauseResult.
Required Test Input	<pre>TEST_METHOD(TestAddNewSynPairResults_nonEmptyExistingResults_success) { ClauseResult cr; string syn1 = "a"; list<int> syn1Results{ 1, 2 }; string syn2 = "b"; list<int> syn2Results{ 4, 5, 6 }; string syn3 = "c"; list<int> syn3Results{ 7, 8, 9 }; cr.updateSynResults(syn1, syn1Results); cr.addNewSynPairResults(syn2, syn2Results, syn3, syn3Results); list<list<int>> actualResults = cr.getAllResults(); list<list<int>> expectedResults{ { 1, 4, 7 }, { 1, 5, 8 }, {1, 6, 9 }, { 2, 4, 7 }, { 2, 5, 8 }, { 2, 6, 9 } }; /***** a b c ----- 1 4 7 1 5 8 1 6 9 2 4 7 2 5 8 2 6 9 *****/</pre>

	<pre> actualResults.sort(); expectedResults.sort(); Assert::IsTrue(actualResults == expectedResults); } TEST_METHOD(TestAddNewSynPairResults_emptyExistingResults_success) { ClauseResult cr; string syn2 = "b"; list<int> syn2Results{ 4, 5, 6 }; string syn3 = "c"; list<int> syn3Results{ 7, 8, 9 }; cr.addNewSynPairResults(syn2, syn2Results, syn3, syn3Results); </pre>
--	--

	<pre> list<list<int>> actualResults = cr.getAllResults(); list<list<int>> expectedResults{ {4, 7}, {5, 8}, {6, 9} }; /***** b c ---- 4 7 5 8 6 9 *****/ actualResults.sort(); expectedResults.sort(); Assert::IsTrue(actualResults == expectedResults); } TEST_METHOD(TestPairWithOldSyn_moreThanExistingResults) { ClauseResult cr; string syn1 = "a"; list<int> syn1Results{ 1, 3, 4 }; string syn2 = "b"; list<int> syn2Results{ 8 }; string syn3 = "c"; list<pair<int, int>> resultPairs; resultPairs.clear(); resultPairs.push_back(pair<int, int>(1, 2)); resultPairs.push_back(pair<int, int>(3, 4)); resultPairs.push_back(pair<int, int>(3, 6)); resultPairs.push_back(pair<int, int>(4, 7)); resultPairs.push_back(pair<int, int>(5, 2)); cr.updateSynResults(syn1, syn1Results); cr.updateSynResults(syn2, syn2Results); cr.pairWithOldSyn(syn1, syn3, resultPairs); list<list<int>> actualResults = cr.getAllResults(); list<list<int>> expectedResults{ { 1, 8, 2 }, { 3, 8, 4 }, { 3, 8, 6 }, { 4, 8, 7 } }; /***** a b c ----- 1 8 2 3 8 4 3 8 6 4 8 7 *****/ actualResults.sort(); expectedResults.sort(); Assert::IsTrue(actualResults == expectedResults); } </pre>
Expected Test Results	true, true, true

5.4 Integration Tests

Integration testing involves the testing of different selected components of the system together. It is assumed that each individual component has been subjected to unit testing.

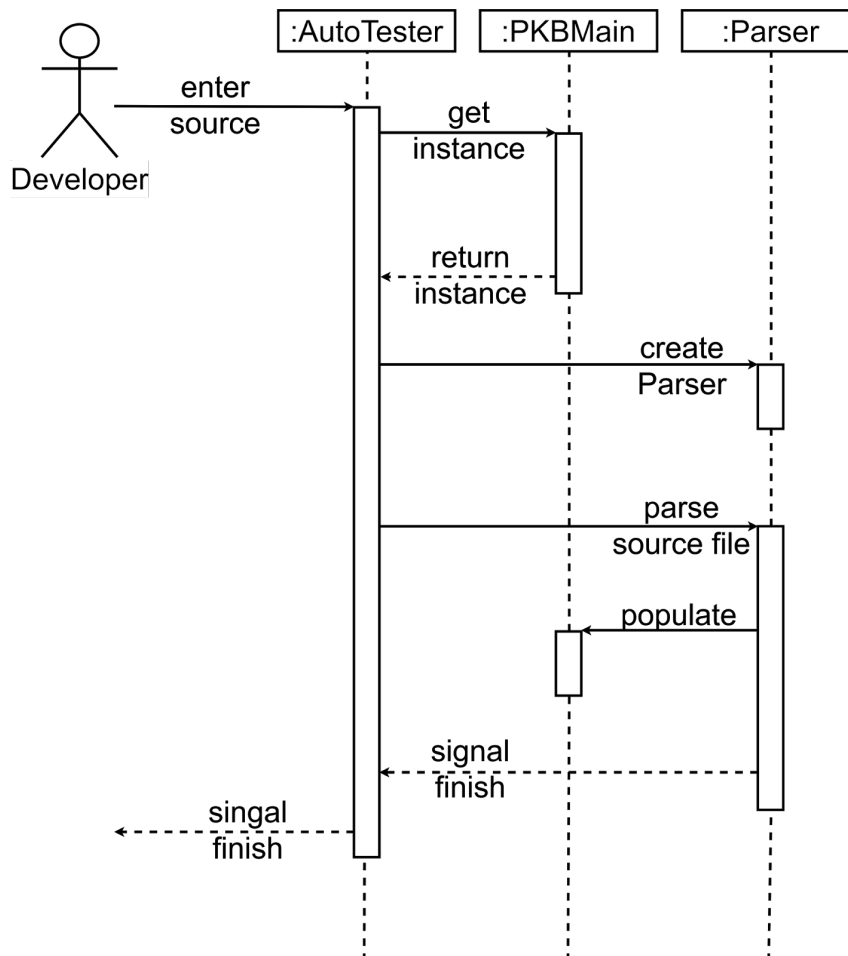


Figure 25: Sequence diagram for AutoTester

5.4.1 Parser - PKB SubSystem

Helper methods are created to create dummy SIMPLE source codes and to destroy them at the end of the integration tests.

After parsing the dummy SIMPLE source codes and populating PKB, actual stored in-

Test Purpose	The interaction between Parser and PKB is one-way, which involves the Parser populating PKB while parsing the SIMPLE source code.
Required Test Input	SIMPLE source code
Expected Test Results	Stored information in PKB

formation is extracted using the API of PKB. This extracted data is then compared to expected ones.

5.4.2 PQL - PKB SubSystem

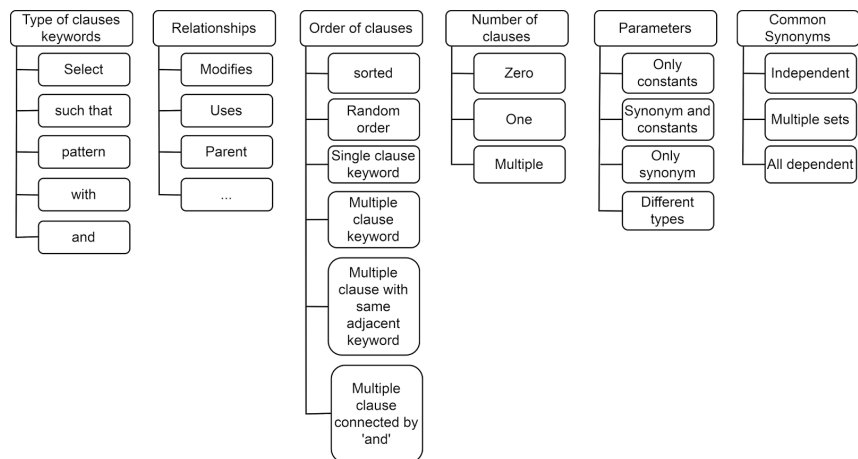
In this example, the components subjected to integration tests are QueryValidator and QueryTree.

5.5 System Tests

System testing involves the testing of all components of the system together. This is carried out with the assistance of the AutoTester. Test cases are designed based on the considerations in Table 9. As mentioned in the previous section, test cases in system testing are controlled and run via a batch file.

<u>Test Purpose</u>	To test if <i>QueryValidator</i> can validate complex query accurately and store the correct data into <i>QueryTree</i> . It specifically targets such that and pattern clauses of different combination.
<u>Required Test Input</u>	<pre> TEST_METHOD (TestValidity_Query_SelectSingleSynonym_Stmt_Parent_Pattern_Assign_And_While_SuchThat_Follows_And_Modifies_Pattern_Valid) { string query; query.append("stmt s1, s2;"); query.append("assign a1, a2;"); query.append("while w1, w2;"); query.append("variable v1, v2;"); query.append("Select s2 such that Parent(s1, s2) pattern a1(v2, _) and w1(\"y\", _) such that Follows(s2, s1) and Modifies(s1, \"x\") pattern w2 (v1, _);"); QueryTree qt; QueryValidator validator = QueryValidator(&qt); Assert::IsTrue(validator.isValidQuery(query)); SelectClause expected = UtilitySelection::makeSelectClause(SELECT_SINGLE, STMT, "s2"); Assert::IsTrue(UtilitySelection::isSameSelectClauseContent(expected, qt.getSelectClause()); vector<SuchThatClause> expectedListStc; vector<PatternClause> expectedListPc; expectedListStc.push_back(UtilitySelection::makeSuchThatClause(PARENT, STMT, "s1", STMT, "s2")); expectedListPc.push_back(UtilitySelection::makePatternClause(ASSIGN_PATTERN, "a1", VARIABLE, "v2", UNDERSCORE, "_")); expectedListPc.push_back(UtilitySelection::makePatternClause(WHILE_PATTERN, "w1", IDENT_WITHQUOTES, "\"y\"", UNDERSCORE, "_")); expectedListStc.push_back(UtilitySelection::makeSuchThatClause(FOLLOWS, STMT, "s2", STMT, "s1")); expectedListStc.push_back(UtilitySelection::makeSuchThatClause(MODIFIES, STMT, "s1", IDENT_WITHQUOTES, "x")); expectedListPc.push_back(UtilitySelection::makePatternClause(WHILE_PATTERN, "w2", VARIABLE, "v1", UNDERSCORE, "_")); Assert::IsTrue(UtilitySelection::AreSameSuchThatClausesContentAsInTree(expectedListStc, qt)); Assert::IsTrue(UtilitySelection::areSamePatternClausesContentAsInTree(expectedListPc, qt)); } </pre>
<u>Expected Test Results</u>	true, true, true, true

Table 7: Integration Testing

**Table 8:** System Testing

Test Purpose	To test if different combinations of clause keywords (such that... and..., such that... such that ..., etc)
Required Test Input	1 - Select SingleSynonym Stmt SuchThat Follows IntInt and Follows IntInt HaveResult stmt s; Select s such that Follows(1, 2) and Follows(2, 3) 1,2,3,4,5,6,7,8 5000 2 - Select SingleSynonym Stmt SuchThat Follows IntInt SuchThat Follows IntInt HaveResult stmt s; Select s such that Follows(1, 2) such that Follows(2, 3) 1,2,3,4,5,6,7,8 5000 ...

Test Purpose	To test if Follows produce the expected result that match SPA requirements.
Required Test Input	1 - Select SingleSynonym Stmt Follows IntSynonym HaveResult stmt s; Select s such that Follows(1, s) 2 5000 2 - Select SingleSynonym Stmt Follows IntSynonym BehindWhile HaveResult stmt s; Select s such that Follows(3, s) 8 5000 3 - Select SingleSynonym Stmt Follows UnderscoreInt HaveResult stmt s; Select s such that Follows(_, 3) 1,2,3,4,5,6,7,8 5000 4 - Select SingleSynonym Stmt Follows UnderscoreInt NoResult stmt s; Select s such that Follows(_, 1) none 5000 5 - Select SingleSynonym Stmt Follows UnderscoreUnderscore HaveResult stmt s; Select s such that Follows(_, _) 1,2,3,4,5,6,7,8 5000 6 - Select SingleSynonym Stmt Follows UnderscoreSynonym HaveResult stmt s;

	Select s such that Follows(_, s) 2,3,5,6,7,8 5000 7 - Select SingleSynonym Stmt Follows SynonymInt HaveResult stmt s; Select s such that Follows(s, 2) 1 5000 8 - Select SingleSynonym Stmt Follows SynonymInt NoResult stmt s; Select s such that Follows(s, 1) none 5000 9 - Select SingleSynonym Stmt Follows SynonymInt BeforeWhile HaveResult stmt s; Select s such that Follows(s, 8) 3 5000 10 - Select SingleSynonym Stmt Follows SynonymUnderscore HaveResult stmt s; Select s such that Follows(s, _) 1,2,3,4,5,6 5000 11 - Select SingleSynonym Stmt Follows SynonymSynonym GetFront HaveResult stmt s1, s2; Select s1 such that Follows(s1, s2) 1,2,3,4,5,6 5000 12 - Select SingleSynonym Stmt Follows SynonymSynonym GetBack HaveResult stmt s1, s2; Select s2 such that Follows(s1, s2) 2,3,5,6,7,8 5000 13 - Select SingleSynonym Stmt Follows SynonymSynonym Same NoResult stmt s; Select s such that Follows(s, s) none 5000 ...
--	--

6 Discussion

6.1 Technical Difficulties

For iteration 2, the technical difficulties were minimal as the team was more used to Visual Studio 2015 and Version Control.

6.2 Project Management

We discussed what went wrong in Iteration 1 and how we could improve it with proper project management. We came up with mini-iterations, and scheduled weekly meetings and coded collaboratively in order to ensure optimal working conditions and motivation for each team member to put their best effort in. This way, we were able to input consistent work.

6.3 Workload and Time Management

Due to the mid term tests for other modules, our project progress was delayed. However, we managed to catch up with the schedule with proper time management schedule. One challenge we faced was trying to design an efficient algorithm to compute star relationships such as Follows*, Parent*, Calls*, Next*, using design extractor and CFG. Furthermore, Query Evaluator had to spend a large amount of time coming up with the suggested APIs for the PKB component so that it can deal with multiple complex clauses. However, we were able to discuss the APIs well in time for the deadline.

7 Future Plan for Iteration 3

We will continue with the current project management plan we have - develop mini iterations and follow them through.

7.1 Features

Parser

For iteration 3, the Parser will be upgraded so that it displays feedback message specific to syntax errors that it detects.

Query Validator

For iteration 3, the QueryValidator will support the validation of relationship Affects and Affects*, as well as the validation of tuple.

A Documentation of Abstract APIs

Given on the next page.