National University of Singapore

CS3201/2 Iteration 3

13th November 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

Our SPA meets all the requirements for Iteration 3 and is fully operational.

1.1 Parser

Our **SPA**, internally referred to as **SPAXI**, is now able to parse a SIMPLE program completely and update PKB with the correct relations.

1.2 PKB

PKB supports **Affects, Affects*** in addition to previous implementations of **Follows, Follows*, Parent, Parent*, Uses, Modifies, Pattern, Calls, Calls*, Next and Next***. It also supports Statement List.

1.3 PQL

The QueryProcessor is able to evaluate PQL Queries with multiple **such that, pattern** and **with** clauses. In addition to **Select BOOLEAN** and **Select synonym**, it can also support selection of **attributes** and **Select <tuple>.**

1.4 Special Achievements

Our testing team automated System Testing, making debugging a lot faster and easier. Further details are given in the testing section.

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			Y
	Implement SIMPLE Parser			Y			
	Implement PKB Tables	Y	Y				
	Implement Design Extractor		Y				
	Implement QueryValidator				Y	Y	Y
	Implement QueryTree						Y
	Implement PQLMain						Y
	Implement 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
	Write documentation	Y	Y	Y	Y	Y	Y
Iteration 2	Extend Parser			Y			
	Design PKB API		Y	Y			Y
	Extend Design Extractor		Y				
	Extend PKBTables	Y	Y				
	Implement Design Patterns			Y	Y		Y
	Extend QueryValidator				Y	Y	
	Refactor QueryTree						Y
	Extend QueryEvaluator					Y	Y
	Implement ClauseResult			Y			
	Refactor PQLMain				Y	Y	
	Implement ResultFormatter					Y	
	Perform Unit Testing		Y	Y	Y	Y	Y
	Peform Integration Testing		Y	Y	Y	Y	
	Perform System Testing		Y	Y	Y	Y	Y
	Write documentation	Y	Y	Y	Y	Y	Y
Iteration 3	Extend Parser			Y			
	Design PKB API		Y				Y
	Extend Design Extractor		Y				
	Extend PKBTables		Y				
	Extend QueryValidator				Y	Y	
	Extend QueryEvaluator					Y	Y
	Build ClauseResult Handler			Y			
	Extend ResultFormatter					Y	
	Implement QueryOptimizer			Y			
	Perform Unit Testing		Y	Y	Y	Y	Y
	Perform Integration Testing		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 3

Mini Iteration 1- Week 9 to end of Week 9 (Sunday)						
Activity	**Ping Chan**	**Zhang Ying**	**Andy**	**Marcus**	**Dash**	**Wen Hao**
Finish Next/Next*		Y				
Implement API for Affects & Affects*		Y				Y
Fix Pattern Bug and extend validator for Affects/Affects*				Y		
Design Query Optimizer	Y					
Perform system testing to fix bugs in Calls/*, Next/*, With				Y		Y
Write SIMPLE source file	Y	Y				
Write System Test Queries	Y	Y	Y	Y	Y	Y
Extend QueryProcessor to handle tuple selection					Y	
Mini Iteration 2- Week 10 to end of Week 10 (Sunday)						
Activity	**Ping Chan**	**Zhang Ying**	**Andy**	**Marcus**	**Dash**	**Wen Hao**
Evaluate Affects & Affects*						Y
Optimise Affects, Affects* & Next* using Cache		Y				
Implement QueryOptimizer	Y					
Write System Test Queries			Y	Y		
Extend QueryProcessor to handle attribute reference					Y	
Perform unit testing	Y	Y		Y	Y	Y
Automate System Testing				Y		
Mini Iteration 3- Week 11 to end of Week 11 (Sunday)						
Activity	**Ping Chan**	**Zhang Ying**	**Andy**	**Marcus**	**Dash**	**Wen Hao**
Extend QueryOptimizer and integrate with Evaluator	Y					Y
Finish Affects*		Y				Y
Update documentation	Y	Y	Y	Y	Y	Y
Debug components	Y	Y		Y	Y	Y
Write SIMPLE source files			Y	Y		
Write System Test Queries			Y	Y		
Port documentation to LaTeX					Y	
Personify AutoTester	Y			Y		
FREEZE FEATURES						
Mini Iteration 4- Week 12 to end of Week 12 (Sunday)						
Activity	**Ping Chan**	**Zhang Ying**	**Andy**	**Marcus**	**Dash**	**Wen Hao**
Write system tests			Y	Y		
Debug system tests	Y	Y		Y	Y	Y
Optimize and debug ClauseResult	Y					
Optimize and debug Validator	Y			Y	Y	
Code clean-up and refactoring	Y	Y		Y	Y	Y
Finalize documentation	Y	Y	Y	Y	Y	Y

3 SPA Design

3.1 Overview

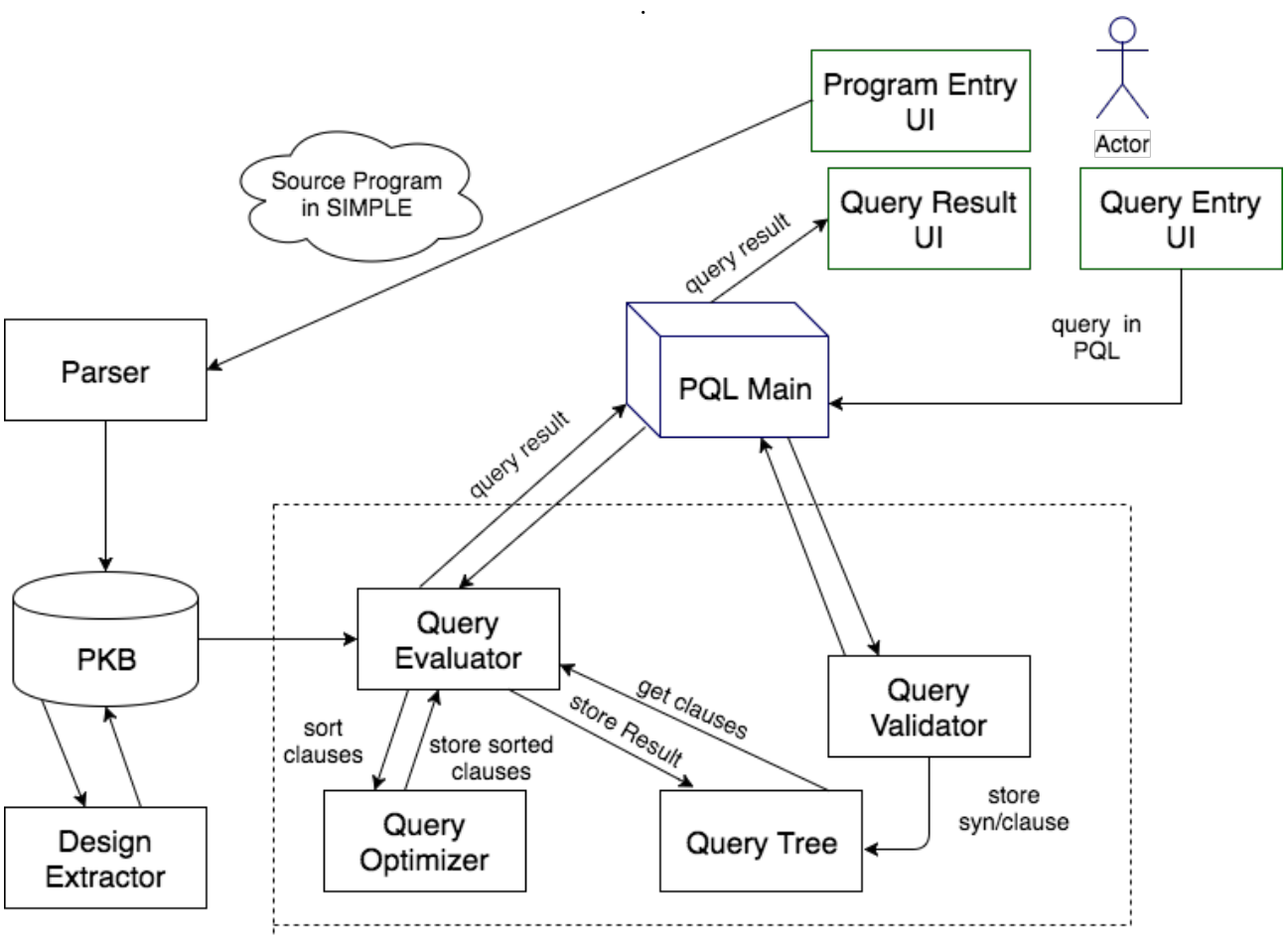


Figure 1: Main components of SPA

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 implemented PQL Main and Query Optimiser. PQL Main controls the flow of all the other components in PQL and decides the sequence during query processing, while Query Optimiser decides the order of evaluation in a query with multiple clauses.

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. The Parser will finish parsing and allow queries to be evaluated only if the SIMPLE source code is syntactically correct.

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 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 {
1    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 is 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.

Do note that our Parser does not allow the LHS of assignment statements to have variables with names that conflict with (if/while/call) keywords.

For example,

```
while = a;
```

is not allowed by our Parser. However, our Parser allows

```
a = while;
```

Uses/Modifies Relation: The Parser adds the Modifies and Uses relationship into PKB as and when the Parser parsers 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.

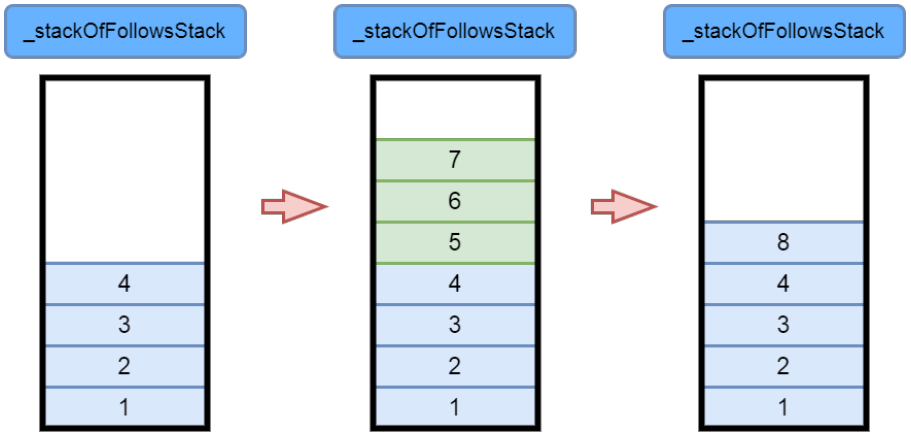


Figure 2: Keeping Track of Follows Relationship

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:

```
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:

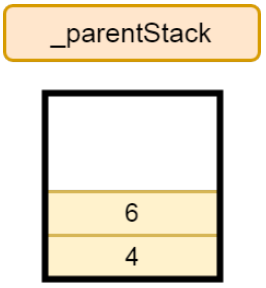


Figure 3: Keeping Track of Transitive Parents

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 the Next relation to PKB:

1. An unordered_set<int> named _prevReachableStmts that stores all the statements that can reach the current statement in one step in the control-flow graph.
2. When parsing the statements inside a while-statement, a variable called whileStmtNumber is used to store the statement number of the while-statement.
3. When parsing the statement lists inside an if-else statement, a variable called ifElseStmtNumber is used to store the statement number of the if-else statement.

Consider the following source code.

```
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 parsing normal statements in the same statement list, such as going from statement 1 to statement 2, Parser will add statement 1 into _prevReachableStmts, and use it to call setNext(_prevReachableStmts, 2) when parsing statement 2. Before moving on to statement 3, _prevReachableStmts is cleared and statement 2 is added into it. This process is repeated if there are more statements in the same statement list without container statements. For the case of container statements, the way Parser sets the Next relation in PKB when parsing while-statements and if-else statements will be described separately for greater clarity.

While-statement: In the above example, when parsing statement 3, Parser will store the statement number in the variable whileStmtNumber. Here, Parser will call PKB's API setNext(3, 4), and then proceeds to parse all the statements in the while-block as per normal. When Parser reaches the last statement in the while-block, i.e. statement 8, it will call set-

Next(8, whileStmtNumber). Here, statements 8 and the whileStmtNumber will be added to the _prevReachableStmts because these two statements have a 1-hop distance to the statement outside the while-block (if any). Upon exiting the while-block, i.e. parsing statement 9, Parser will call setNext(s, 9) for all s that are present in _prevReachableStmts, and _prevReachableStmts is cleared.

if-else statement: When parsing statement 5, Parser will store the statement number of the if-else statement in an integer variable called ifElseStmtNumber. Parser then calls setNext(ifElseStmtNumber, 6). Before exiting the if-block, the last statement number in the if-block is stored in a variable called ifBlockExitPoint. When parsing the first statement in the else-block, Parser will call setNext(ifElseStmtNumber, 7). Before exiting the else-block, _prevReachableStmts is cleared and the last statement of the if- and else- blocks are added to _prevReachableStmts. Upon exiting the if-else block, setNext(s, 8) is called for all s that are present in _prevReachableStmts, and _prevReachableStmts is cleared.

3.2.2 PKB

Program Knowledge Base (PKB) stores the different program design abstractions. 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 allows quick access of the data. Different class table files were made according to the different design entities. This allows easier additions of new design entities and eases code extension as well as reducing modifications required.

PKB contains the following components:

- PKB Main
- Procedure Index Table
- Variable Index Table
- Constant Table
- Uses Table
- Modifies Table
- Pattern Table
- Follows/Follows* Table
- Parents/Parent* Table
- Calls/Calls* Table
- Next Table

PKB-Main: PKB-Main controls and stores all the tables in the entire PKB component. It is used by SPA front-end Parser and Query Processor components in order to reduce coupling. The APIs of the PKB-Main play 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 in a similar fashion 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'

Constant Table

The Constant Table stores all the constants added by the Parser. The statement number is stored in the key and the constants are stored as the value. The table will be accessed when QueryEvaluator calls 'Select c'.

Its structure is as follows:

ConstantTable.h

- unordered_map<int, list<int> > constantTableMap

Uses Table
The Uses Table stores all the Uses relationships extracted by the Parser in the form of 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 are 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 keys, and statement numbers as values. UsesTableVar 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 internally converted 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.
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.

For example, 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 structure of the table as follows:

FollowsTable.h

- 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 is parsed by the Parser, it adds the statement number as a key, and the statements that 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 QueryEvaluator.

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 QueryEvaluator.

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. They also store 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
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.

The structure is as follows:

NextTable.h

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

Next*
As Next* can only be computed on the fly whenever a query comes in, it will not be computed in the design extractor. It is computed in next table using the already stored information in NextTable. The computation of Next* is using a breadth-first iteration on all Next relations depending on the query type. Only the result of a Next* query with two synonym would be cached as it will contain all Next* relations for future clauses.

Cache
The cache stores information regarding Next*, Affects and Affects* that was computed during the evaluation of the given query. The purpose of the cache is to avoid recomputing the aforementioned relations if there are multiple same such-that clauses.

For example, in this query

```
Select a such that Affects*(a, a1) and Affects*(1, 17)
```

PKB would compute all the Affects* relations(for Affects*(a, a1)) and store them in the cache such that the required information for the next clause(Affect*(1, 17)) is retrieved in constant time. The same could be said for Next* and Affects. The cache is cleared after the evaluation of each query.

The structure of the Cache is as follows:

- unordered_map<int, list<int> > nextStarMapList/nextStarMapListReverse
- unordered_map<int, list<int> > affectsMapList/affectsMapListReverse
- unordered_map<int, list<int> > affectsStarMapList/affectsStarMapListReverse
- unordered_map<int, unordered_set<int> > nextStarMap
- unordered_map<int, unordered_set<int> > affectsMap
- unordered_map<int, unordered_set<int> > affectsStarMap

Computation of Affects and Affects*

To compute Affects and Affects*, we implement a one pass method to extract all relations. This method contains an essential component: the *latestModifiedMap*. This map contains the mapping from a variable to the statements that directly(for Affects) or indirectly(for Affects*) modify it. The method also makes use of a *whileMapStack* which keeps track of the while statement and its respective *latestModifiedMap*, as well as an *ifMapStack* that keeps track of the if statement and its respective *latestModifiedMaps* for the if-else branch. The *ifMapStack* is a stack of *IfStmt* objects which will be described in detail under *IfStmt*.

- unordered_map<int, unordered_set<int> > latestModifiedMap
- stack<pair<int, latestModifiedMap> > whileMapStack
- stack<IfStmt> ifMapStack

IfStmt

The IfStmt object contains the necessary information required for processing an if statement. It will store the statement number of the if statement(stmt), the first statement of the if block(startIf), the last statement of the if block(endIf), the first statement of the else block(startElse), the last statement of the else block(endElse), whether the else block has been visited or not(visitedElse), the statement following the if statement (which is 0 is the if statement is the last) in the procedure(afterIf), the latestModifiedMap after iterating the if block(ifModifiedMap) and the latestModifiedMap after iterating the else block(elseModifiedMap). Once the if and else block have been iterated, the ifModifiedMap and elseModifiedMap will be merged and replace the current latestModifiedMap.

- int stmt
- int startIf
- int endIf
- int startElse
- int endElse
- bool visitedElse
- int afterIf
- latestModifiedMap ifModifiedMap
- latestModifiedMap elseModifiedMap

The PKB computes Affects/* with the one-pass method. It will apply the extraction method starting from the first statement of each procedure.

Algorithm for extracting all affects relation

For every statement in the procedure:

- If it is an assignment
  - Check if any variables used in the statement is in the latestModifiedMap and add affects relation to cache if it is
- If it is an assignment or call
  - Remove the variables modified in current statement from latestModifiedMap
  - Insert the modified variable into latestModifiedMap in the form <variable, statement>
- If it is a while statement
  - Insert into the whileMapStack in the form <statement, latestModifiedMap>
  - Iterate the while loop until there are no new additions
  - Merge the new latestModifiedMap with the oldModifiedMap in the whileMapStack
- If it is an if statement
  - Insert into the ifMapStack in the form <statement, startIf, endIf, startElse, endElse, *false*, afterIf, latestModifiedMap, latestModifiedMap>
  - We iterate through the if CFG then update its latestModifiedMap, then iterate through the else CFG and update its latestModifiedMap before merging both if and else's latestModifiedMap together.
- Proceed to next statement until procedure ends

Restart the algorithm for a new procedure

Algorithm for extracting all affects* relations

The algorithm for extracting all affects* in one-pass is very similar to extracting all affects relations. The only additional check is for an assignment statement. (marked with *)

For every statement in the procedure:

- If it is an assignment
  - Check if any variables used in the statement is in the latestModifiedMap and add affects* relation to cache if it is
- If it is a call
  - Remove the variables modified in current statement from latestModifiedMap
  - Insert the modified variable into latestModifiedMap
- If it is an assignment
  - Remove the variables modified in current statement from latestModifiedMap only if it does not use the variable that it modifies (i.e. x = x;) *
  - Insert the modified variable into latestModifiedMap in the form <variable, statement>
- If it is a while statement
  - Insert into the whileMapStack in the form <statement, latestModifiedMap>
  - Iterate the while loop until there are no new additions
  - Merge the new latestModifiedMap with the oldModifiedMap in the whileMapStack
- If it is an if statement
  - Insert into the ifMapStack in the form <statement, startIf, endIf, startElse, endElse, *false*, afterIf, latestModifiedMap, latestModifiedMap>
  - We iterate through the if CFG then update its latestModifiedMap, then iterate through the else CFG and update its latestModifiedMap before merging both if and else's latestModifiedMap together.
- Proceed to next statement until procedure ends

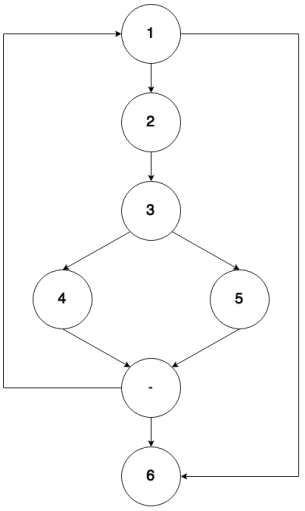
Restart the algorithm for a new procedure

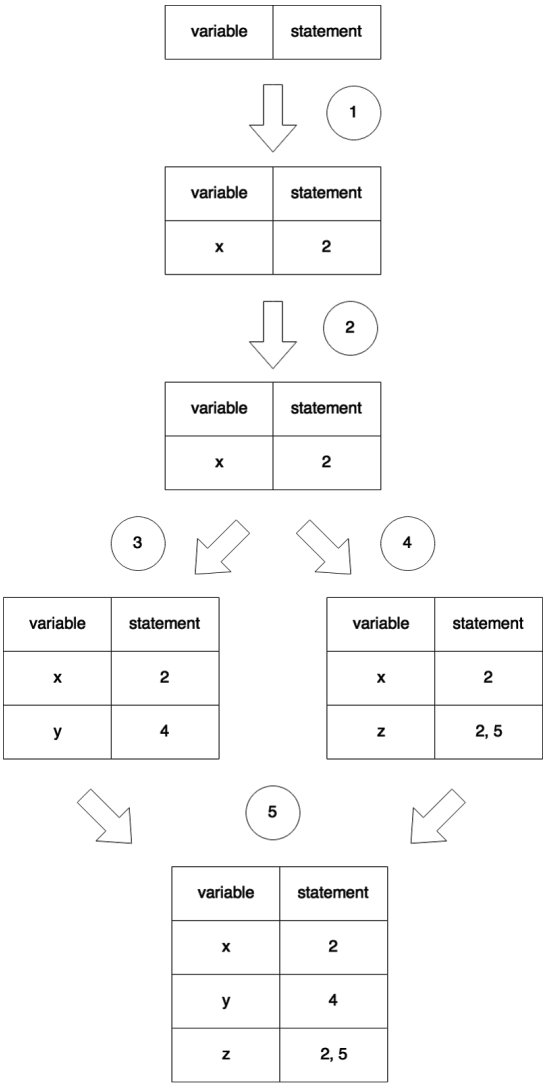
Illustrated Example

As the extraction of affects and affects* relations are relatively similar, for the illustrated example, we will be going through the extraction of affect* relations.

Given the source code and CFG below:

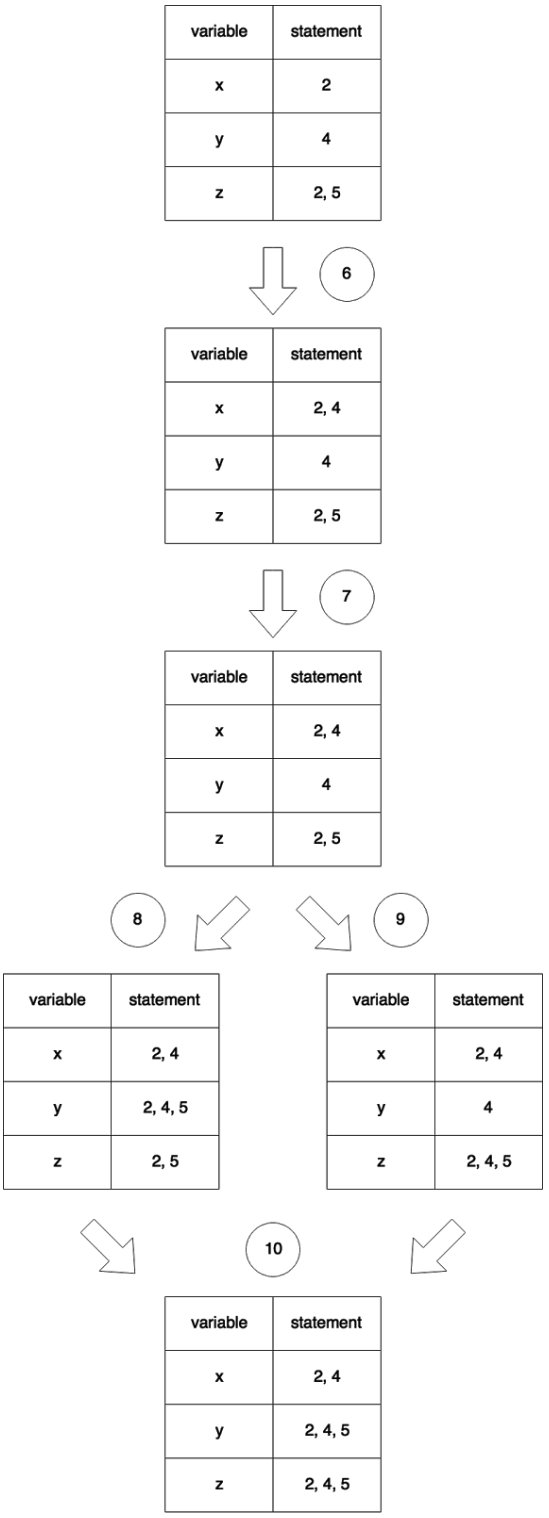
```
procedure Example {
1    while x {
2        x = y;
3        if x then {
4            y = z; }
         else {
5            z = x; }
     }
6    a = x;
}
```





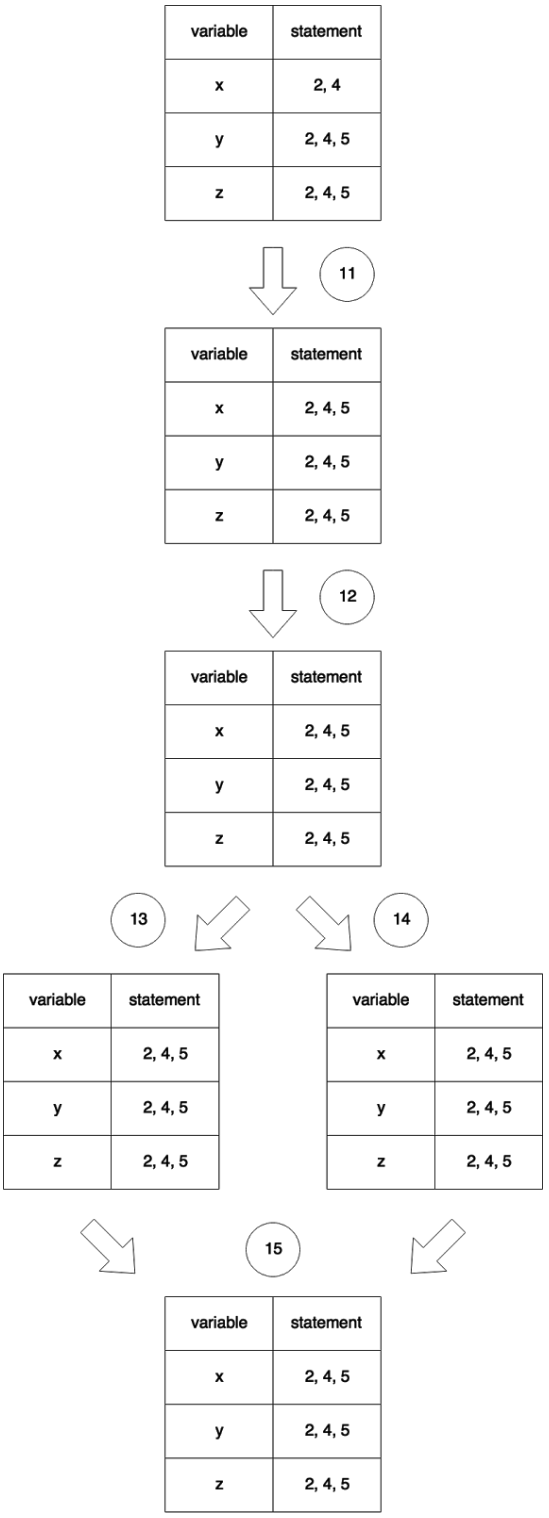
At the start, the **latestModifiedMap** is empty. As **stmt 1** is a while loop, we push the empty **latestModifiedMap** into the **whileMapStack**.

1. Stmt 2 uses **y**, but the **var y** is not in the latestModifiedMap, so we continue and put the **relation x** modified by stmt 2. (directly/indirectly) into the map.
2. As this is an if-statement, we branch out into arrows 3 and 4, making a copy of the current map as we iterate the if-else block.
3. We iterate the if block. Since **y uses z**,which does not exist in the **latestModifiedMap**, we simply put the relation **y** modified by stmt 4 (directly/indirectly) into the map.
4. We now iterate the else block. Since stmt 5 uses **x** and **z** is affected by **x** which exists in the **latestModifiedMap**, we add the statements that modify z (directly/indirectly) into the map for var z. We also **add the affects* relation (2, 5)** into the cache. We then put the current statement, 5, as modifying z into the mapping from z.
5. At step 5, we have already finished iterating the if-else block and now merge the 2 separate **latestModifiedMaps** together. Since the merged map is different from the map we started with (the map on the **whileMapStack**), we have to continue to iterate the while loop until there are no new changes.



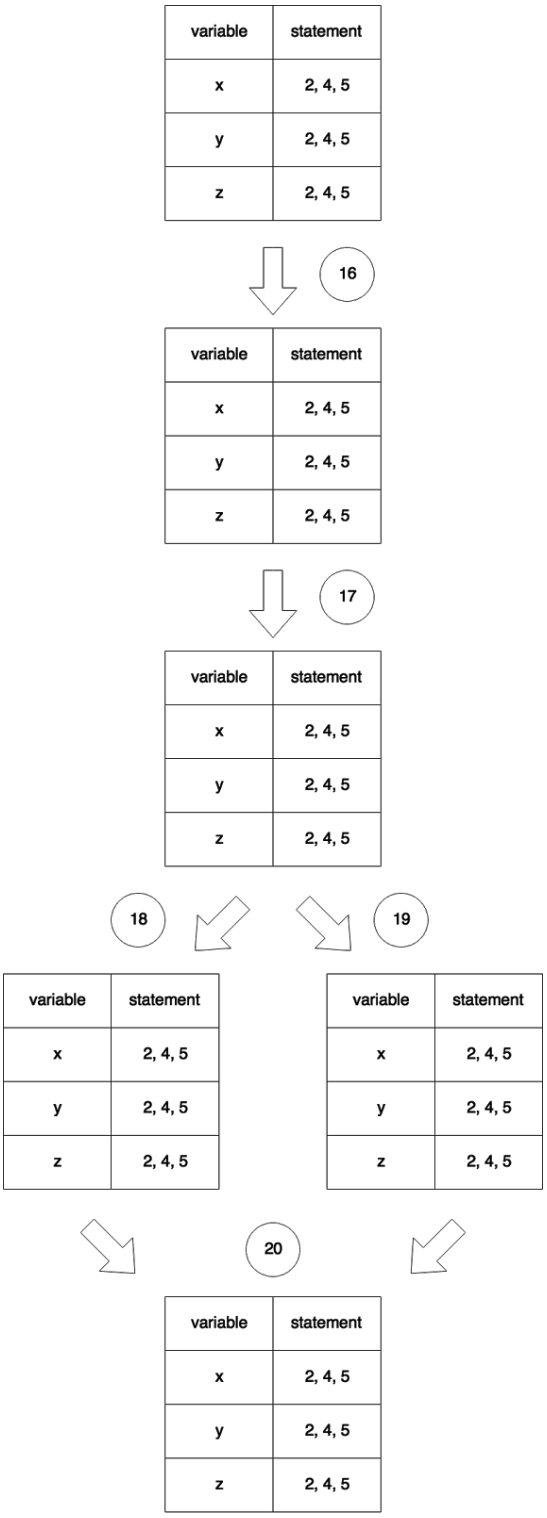
We iterate the while loop again with the latestModifiedMap generated from merging at step 5.

6. We proceed into the while loop at statement 2. Since statement 2 uses y and y exists in the map, we **add the affects* relation (4, 2)** into the cache. At the same time, the statement 4 will be added to the set mapped from variable x.
7. We branch out at statement 3 is an if statement, making a copy of the latestModifiedMap for our use in the if and else branch.
8. Since statement 4 uses z and z exists in the map, we **add the affects* relation (2, 4) and (5, 4)** into the cache. We then also add statements 2 and 5 into the set being mapped from variable y.
9. Since statement 5 uses x and x exists in the map, we **add the affects* relations (2, 5) and (4, 5)** into the cache. We then also add statements 2 and 4 into the set being mapped from variable z.
10. As we have finished iterating through the if and else block, we merge the two maps from each path. We then compare the merged map with the map on the whileMapStack, in this was is the map before step 6. Since the maps are still different, this means that we have to iterate the while loop again.



We iterate the while loop again with the new latestModifiedMap generated from the merging at step 10.

11. We go on to statement 2 where y is being used. We then **add the affects* relations (2, 2), (4, 2), (5, 2)** into the cache. We then add the additional statements that directly/indirectly modified x into the map.
12. Once again, as statement 3 is an if statement, we make a copy of the latestModifiedMap and branch them out when we iterate the if and else block.
13. As statement 4 uses variable z, we **add the affects* relations (2, 4), (4, 4) and (5, 4)** into the cache. Then we update the set mapped from y accordingly.
14. As statement 5 uses variable x, we **add the affects* relations (2, 5), (4, 5) and (5, 5)** into the cache. Then we update the set mapped from z accordingly.
15. We merge the 2 maps from if branch and else branch at this point. We then compare the map with the map we entered the loop with. As they were changes made, we have to iterate the while loop one more time.



We iterate the while loop again with the new latestModifiedMap generated from the merging at step 15.

16. We go on to statement 2 where y is being used. We then **add the affects* relations (2, 2), (4, 2), (5, 2)** into the cache. We then add the additional statements that directly/indirectly modified x into the map.
17. Once again, as statement 3 is an if statement, we make a copy of the latestModifiedMap and branch them out when we iterate the if and else block.
18. As statement 4 uses variable z, we **add the affects* relations (2, 4), (4, 4) and (5, 4)** into the cache. Then we update the set mapped from y accordingly.
19. As statement 5 uses variable x, we **add the affects* relations (2, 5), (4, 5) and (5, 5)** into the cache. Then we update the set mapped from z accordingly.
20. We merge the 2 maps from if branch and else branch at this point. We then compare the map with the map we entered the loop with. Since the maps are now the same which means that we have captured all relations inside the while loop and updated the map accordingly, we can now exit the while loop and proceed to the statement after it, statement 6.

variable	statement
x	2, 4, 5
y	2, 4, 5
z	2, 4, 5

21

variable	statement
x	2, 4, 5
y	2, 4, 5
z	2, 4, 5
a	2, 4, 5, 6

21. At statement 6, since 6 uses x, we **add the affects* relations (2,6), (4, 6), (5, 6)** into the cache. At the same time, we update the variable modified, a, with the statements that directly/indirectly modifies the variables used.

With this, we have finished extracting all the affects* relations in procedure Example. If there are more procedures, we will continue to do so with them. As noted above, there might be instances where we add duplicate affects* relations into our cache. The cache will be able to detect duplicates and remove them accordingly in order to be memory efficient.

The final **Affects* relations** found in the cache are as follows:
(2, 2), (2, 4), (2, 5), (2, 6), (4, 2), (4, 4), (4, 5), (4, 6), (5, 2), (5, 4), (5, 5), (5, 6).

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 inter-procedural 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 algorithm. 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

ParentToChildTable

Parent	Child
1	2
2	3
3	4

ParentToChildStarTable

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

Figure 4: Parent to Child/ChildStar Table

computation of Parent* relations from the Parent relations table given in 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

ChildToParentTable

Child	Parent
2	1
3	2
4	3

ChildToParentStarTable

Child	Parent
2	1
3	2, 1
4	3, 2, 1

Figure 5: Child to Parent/ParentStar Table

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 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.

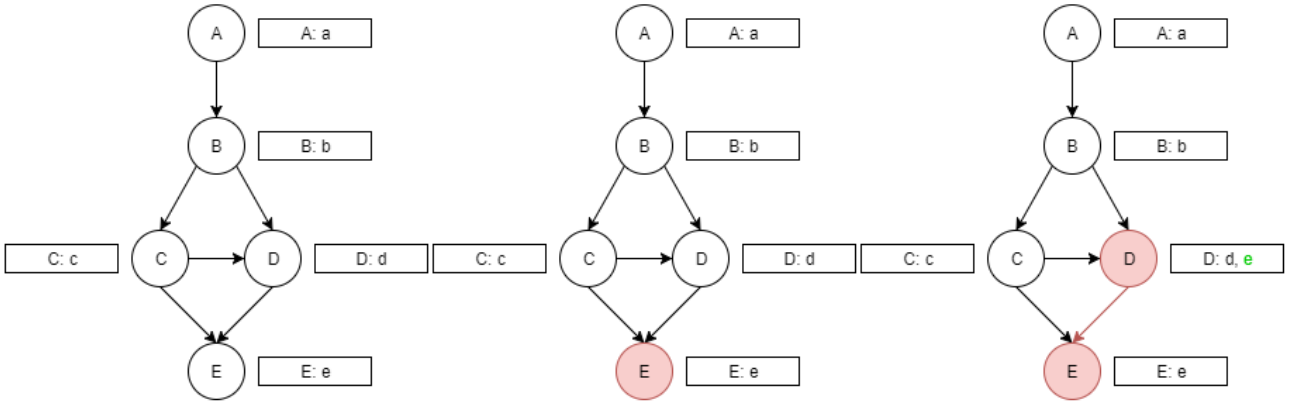


Figure 6: Call Traversal from Proc A

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 it's 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).

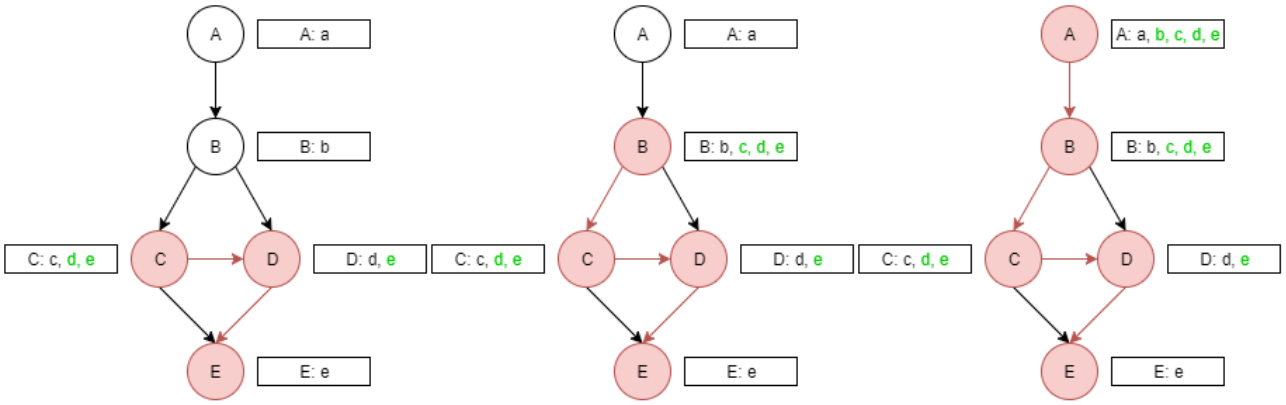


Figure 7: Call Traversal from Proc A

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

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

Statement	Variables
1	a, b, c
2	a, b, c
3	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.9 shows the component diagram of the PQLMain and its sub-components.

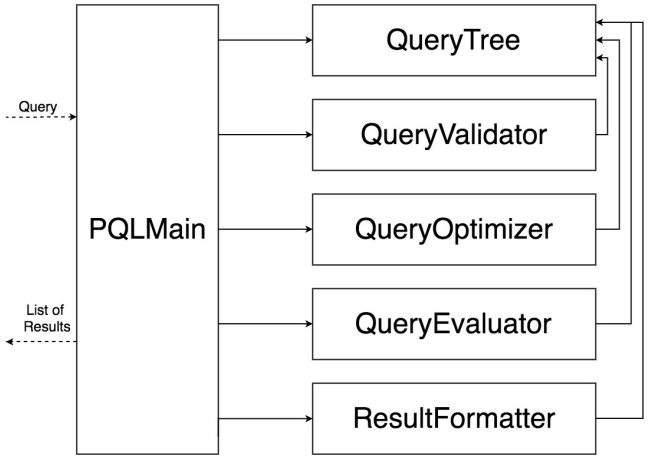


Figure 9: PQLMain Facade Component diagram

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 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 semicolons 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.

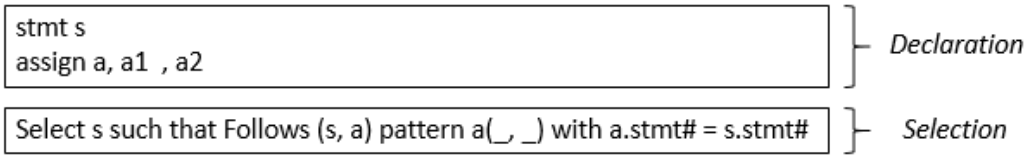


Figure 10: State of query after dissection

2. The QueryValidator uses regex to extract and validate the declaration entities and synonyms. The QueryValidator will split the declaration into the entity token and synonym token. The first sub-token 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. Validation proceeds similarly for synonym tokens. If the entire declaration is valid, the entire declaration is stored in the QueryTree. Do note that we do not allow duplicate declarations.

   For example,

   ```
   stmt s; stmt s;
   Select s

   stmt s, s;
   Select s
   ```

   are considered invalid.

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 Selection-

Validator. 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.

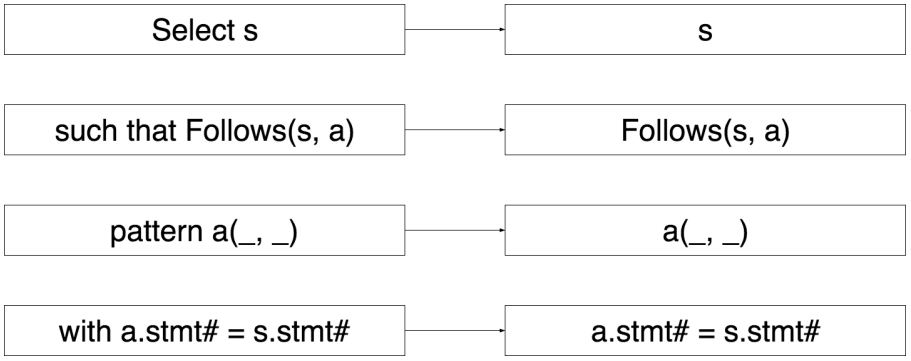


Figure 11: QueryValidator Selection: State of clause before SelectionValidator passes it to the respective handlers

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.

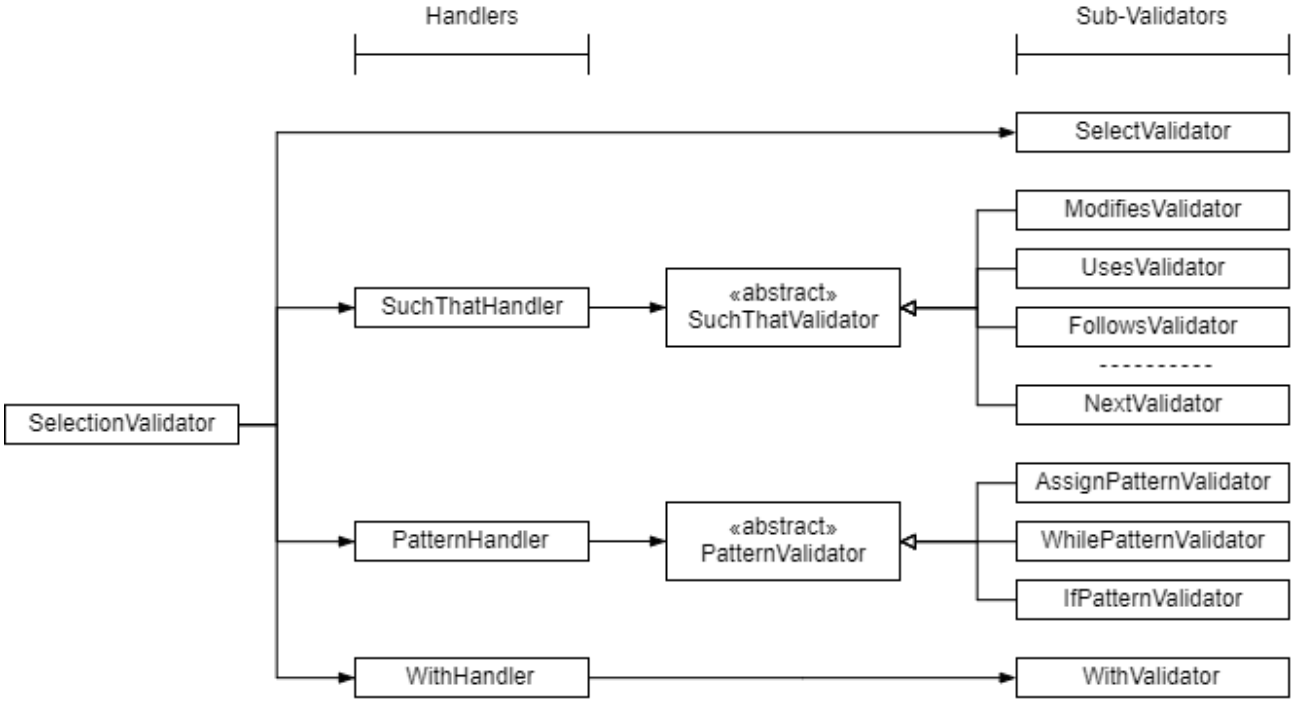


Figure 12: QueryValidator Handler: Possible path taken

Each sub-validator extracts the argument and enquires the QueryTree for the 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 Entity	
STMT	0
ASSIGN	1
WHILE	2
IF	3
PROG_LINE	4
CALL	5
PROCEDURE	6
VARIABLE	7
INTEGER	8
UNDERSCORE	9
IDENT_WITHQUOTES	10
EXPRESSION_SPEC_PARTIAL	11
EXPRESSION_SPEC_EXACT	12
CONSTANT	13
STMTLIST	14

enum Relationship	
MODIFIES	0
USES	1
PARENT	2
PARENTSTAR	3
FOLLOWS	4
FOLLOWSSTAR	5
CALLS	6
CALLSSTAR	7
NEXT	8
NEXTSTAR	9
AFFECTS	10
AFFECTSSTAR	11

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_ATTIBUTE	7
INTEGER_ATTRIBUTE	8
IDENT_WITH_QUOTES_ATTRIBUTE	9
PROG_LINE_ATTRIBUTE	10

enum PatternType	
ASSIGN_PATTERN	0
WHILE_PATTERN	1
IF_PATTERN	2

Table 1: Enums

Do note that due to ambiguity in the detection of **Select BOOLEAN** in an invalid query, we will return *false* whenever we detect BOOLEAN in a query.

For example:

```
Select BOOLEAN hello
```

returns false.

For Iteration 3, we have extended our SPA to validate **Select tuples** and Select synonyms with **attributes**.

For attribute validation, we use regex to split the synonym into synonym and attribute. We first validate the synonym syntactically. Then we proceed with semantic validation for its corresponding attribute type. Here, since CALL entity can have two attributes - procName and stmt# - we also store an additional flag in the QueryTree to let the ResultFormatter know whether to get the string mapping from PKB for CALL when projecting results to the UI.

For tuple validation, we extract each tuple argument and proceed to validate it as we would validate each synonym, and store it in the QueryTree.

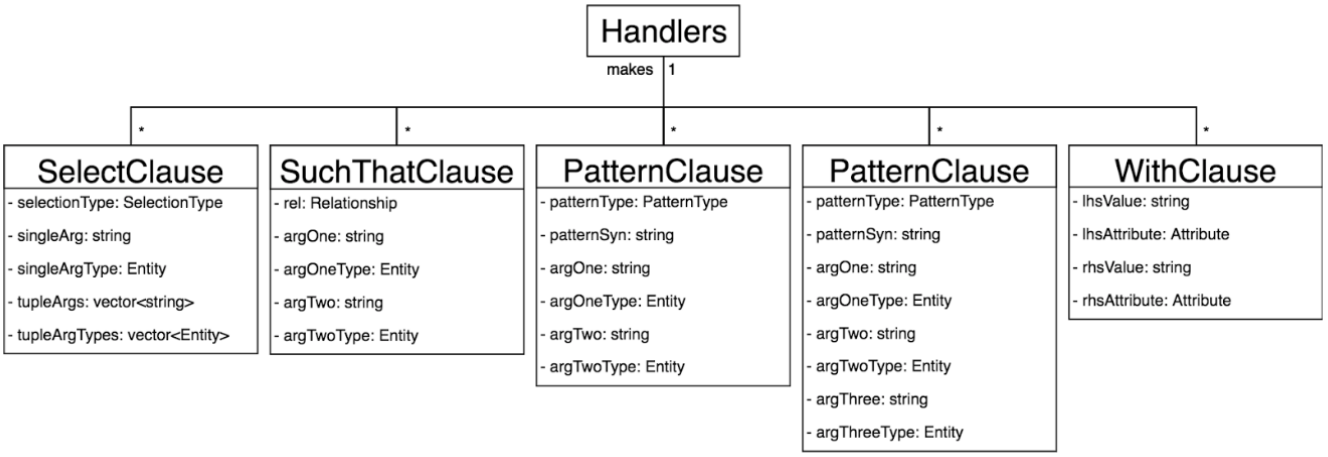


Figure 13: Query Validation Handler

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.

The decision making at each stage is shown below.

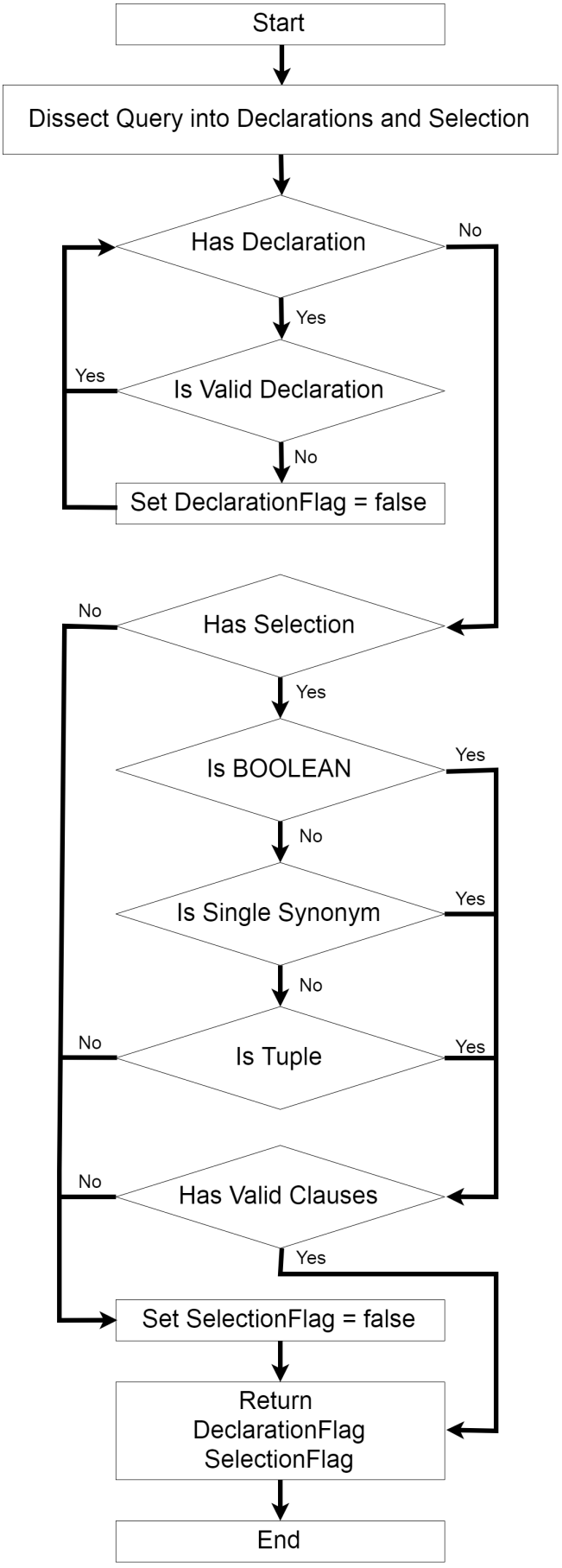


Figure 14: Validator Flowchart

3.2.6 Query Optimiser

The QueryOptimiser is designed to pre-process the clauses in a query and perform optimisation such that inefficient evaluation of the query clauses is avoided and near optimal evaluation time can be achieved by the QueryEvaluator. After being created by the QueryValidator, the QueryTree is passed to the QueryOptimiser before it is passed to the QueryEvaluator for evaluation. First and foremost, QueryOptimiser will perform some basic pre-processing. Duplicate clauses are removed, and when there are with-clauses with ident, overwriting of synonyms is also done. For example:

```
procedure p; stmt s;
Select s such that Uses(p,"x") with p.procName = "funProcedure"
```
Uses-clause will be modified to Users(funProcedure, "x") and the with-clause will be removed from the queryTree.

The QueryOptimiser then groups and reorders the QueryTree based on the following heuristics:

1. Avoid cross-products with synonyms not already present in intermediate table
2. The more synonyms present in the intermediate result table, the larger it is likely to become
3. The larger the results of a clause, the more time-consuming cross-product operation it'll likely cause in the future
4. With-clauses and pattern-clauses yield very small sets of results, therefore they should be prioritised
5. Compute Next, Next*, Affects and Affects* at the end
6. Overwriting synonyms whenever possible (using With clauses with ident)
7. Select-clause can result in very big intermediate results if the query is selecting tuple with many synonyms that are not present in other clauses

Most of the heuristics above aim to minimise number of cross-products, minimise the intermediate result size (to lower the cost of cross-product), and evaluate clauses that need to be computed during runtime only after the cheaper clauses are evaluated.

One strategy used to extract all the clauses in the QueryTree and split them into various clause groups is based on the common synonyms they have. The grouping is done such that all synonyms only belong to only one group, and each synonyms in a clause group is "linked" to every other synonyms in the same clause group by at least one clause. A synonym is said to be "linked" to another synonym if both of them exist as arguments in a clause, e.g. "a1" is linked to "v1" by the clause Modifies(a1, v1).

In addition, the Optimiser also stores a table that assigns an estimated cost to each type of clause. SpaXI's Optimiser estimates the cost of clauses based on statistical estimation and heuristics, as well as the size of the result.

Cost = Number of Computations (PKB) + Result Size

The result size of a clause is as important as computation time because the size of the result will influence the time taken of cross-product in the intermediate result.

This table lists the summary of the costs allocated to each clauses, sorted in ascending order, for a source code with 500 statements.

Clause Type	**Cost**	**Worst-case time comlpexity**	**Result size**
WITH_ANY_ARGS	0	0	0
FOLLOWS_BOOLEAN	1	$O(1)$	0
NEXT_BOOLEAN	1	$O(1)$	0
CALLS_BOOLEAN	1	$O(1)$	0
PARENT_BOOLEAN	1	$O(1)$	0
CALLS_STAR_BOOLEAN	1	$O(1)$	0
PARENT_STAR_BOOLEAN	1	$O(1)$	0
MODIFIES_BOOLEAN	1	$O(1)$	0
USES_BOOLEAN	1	$O(1)$	0
FOLLOWS_STAR_BOOLEAN	1	$O(1)$	0
FOLLOWS_1ARG	2	$O(1)$	1
NEXT_1ARG	2	$O(1)$	1
CALLS_1ARG	11	$O(1)$	10
CALLS_STAR_1ARG	31	$O(1)$	30
CALLS_2ARGS	121	$O(1)$	120
CALLS_STAR_2ARGS	151	$O(1)$	150
SELECT_1ARG	501	$O(1)$	500
PARENT_1ARG	501	$O(N)$	1
MODIFIES_1ARG	525	$O(N)$	25
USES_1ARG	550	$O(N)$	50
PATTERN_1ARG	750	$O(N)$	250
PARENT_STAR_1ARG	800	$O(N)$	300
PATTERN_2ARGS	1000	$O(N)$	500
FOLLOWS_STAR_1ARG	1000	$O(N)$	500
PARENT_2ARGS	1000	$O(N)$	500
FOLLOWS_2ARGS	1000	$O(N)$	500
NEXT_2ARGS	1000	$O(N)$	500
NEXT_STAR_BOOLEAN	1500	$O(V+E)$	0
NEXT_STAR_1ARG	2000	$O(V+E)$	500
PARENT_STAR_2ARGS	2000	$O(N)$	1500
MODIFIES_2ARGS	5500	$O(N)$	5000
USES_2ARGS	15500	$O(N)$	15000
FOLLOWS_STAR_2ARGS	50500	$O(N)$	50000
AFFECTS_BOOLEAN	90000	$O(2*(V+E))$	0
AFFECTS_1ARG	90040	$O(2*(V+E))$	40
NEXT_STAR_2ARGS	875000	$O(V*(V+E))$	125000
AFFECTS_STAR_BOOLEAN	20833250	$N(N+1)(N+2)/6$	0
AFFECTS_STAR_1ARG	20833350	$N(N+1)(N+2)/6$	100
AFFECTS_2ARGS	20843250	$N(N+1)(N+2)/6$	10000
AFFECTS_STAR_2ARGS	20953250	$N(N+1)(N+2)/6$	120000

SELECT_TUPLE	INF	–	INF

The following table gives a more comprehensive explanation of how the PKB computation cost is calculated.

Clause Type	**PKB Computation Time** (for 500 LoC SIMPLE source)
WITH_ANY_ARGS	Always put as first clause based on the heuristic reasoning that with-clauses generally result in very small result size. The only exception is when there are two synonyms which are stmt.stmt# and prog_line.progline#. But this happens very rarely and SpaXI ignores it.
FOLLOWS_BOOLEAN NEXT_BOOLEAN CALLS_BOOLEAN PARENT_BOOLEAN CALLS_STAR_BOOLEAN PARENT_STAR_BOOLEAN MODIFIES_BOOLEAN USES_BOOLEAN FOLLOWS_STAR_BOOLEAN FOLLOWS_1ARG NEXT_1ARG CALLS_1ARG CALLS_STAR_1ARG CALLS_2ARGS CALLS_STAR_2ARGS SELECT_1ARG	Constant time as these data are stored using unordered_map in PKB, $O(1)$
NEXT_STAR_BOOLEAN NEXT_STAR_1ARG	Breath-first search is performed on the CFG. $O(V + E)$, where V and E are the number of vertices and edges in the CFG respectively. Est: $V = 500$; $E = 2 * V$
PARENT_STAR_2ARGS MODIFIES_2ARGS USES_2ARGS FOLLOWS_STAR_2ARGS	$O(N)$, N = max num of stmts (sieving) Est: $N = 500$

AFFECTS_BOOLEAN AFFECTS_1ARG	2 times of Next* computation time BFS is performed on CFG. For each vertex, check Uses relation. Complexity is $O(2 * (V + E)) * (avgnumofusedvariables)$, where V is the number of vertices and E is the number of edges in the CFG. Est: $V = 500; E = 2 * V$; avg used number of variable = 30
NEXT_STAR_2ARGS	$O(N)$, N = max num of stmts (sieving) Est: N = 500
AFFECTS_BOOLEAN AFFECTS_1ARG	Num of nodes x Complexity of Next* BFS: $O(V * (V + E))$ Est: $V = 500; E = 2 * V$
AFFECTS_STAR_BOOLEAN AFFECTS_STAR_1ARG AFFECTS_2ARGS AFFECTS_STAR_2ARGS	Num of nodes x Complexity of Next* BFS: $O(V * (V + E))$ Est: $V = 500; E = 2 * V$
SELECT_TUPLE	$O(1 * tupleSize)$

Clauses within all clause groups are then sorted based on the costs of each clause and chained such that clauses with two synonyms have one of the synonyms already evaluated before their own evaluation (unless all the clauses in a clause group have two synonyms). The sorted clauses are then enqueued in a std::queue<Clause>. Clause groups are also sorted based on the total cost of the clauses in the clause groups, and enqueued in a std::queue<ClauseGroup> in a similar way.
For SpaXI, because the intermediate clause result object performs frequent pruning and is able to store non-selected synonyms for as long as they are needed, the Optimiser will not need to consider whether synonyms are "selected" when sorting clauses and clause groups.

By following the above strategies, if we have n number of clauses, they will usually be grouped in the following way:

1. **Group 0:** Clauses that return boolean results (e.g Follows(1,2), Uses(1,"x"))
2. **Group 0 to Group n-2:** 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)
3. **Last Group:** Contains only one new Select Clause created in Optimiser that only contain the "selected" synonyms that are not present in the other clauses. If all the "selected" synonyms are present in other clauses, this group will not be created.

After the QueryOptimiser is done with sorting the clauses as outlined above, it passes the queue<ClauseGroup> to a ClauseGroupManager, which abstract away the processes and operations of managing the clause groups, and merges the results of each clause group. This gives greater flexibility in clause group management and makes the system more extensible and maintainable. The QueryEvaluator asks for the next ClauseGroup from the ClauseGroupManager and processes each ClauseGroup one by one. A ClauseResult (intermediate result) will be produced for each ClauseGroup, which will be passed to the ClauseGroupManager to be merged with the ClauseResult of the previously evaluated clause groups. Internally, the ClauseGroupManager will only merge the synonyms that are selected in order to minimise the impact of cross-product.
Another thing to notice is that, in the midst of evaluating a ClauseGroup, if a synonym is not selected in the Select clause in a query and is no longer present in the clauses that have not been evaluated yet, then the possible results of the synonym no longer need to be stored and can be pruned from the intermediate result. This will help to reduce the size of the intermediate result significantly. In order to do this, each ClauseGroup uses an unordered_set<string> with the name _remainingSynonyms to store the synonyms that are present in each clauses in the ClauseGroup. It also stores the synonyms that are selected in the Select clause. Whenever a clause has been evaluated, the ClauseGroup will update _remainingSynonyms such that it only contains the synonyms that are in the remaining clauses that are yet to be evaluated. The _remainingSynonyms variable as well as the selected synonyms can then be used to prune the intermediate result object ClauseResult.

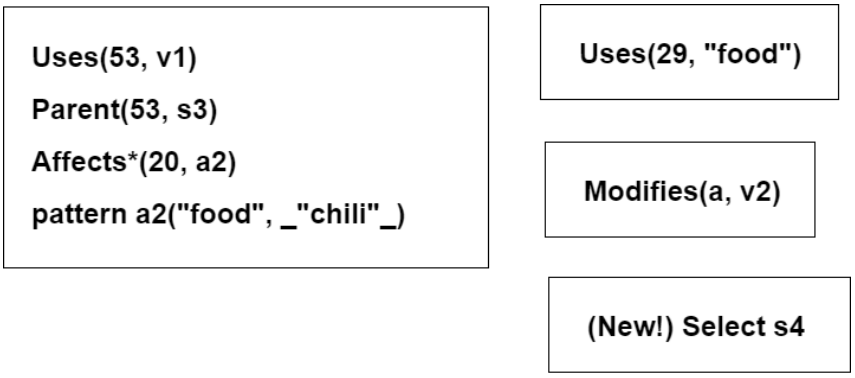
For example, consider the following query:

```
assign a, a1, a2; stmt s1, s2, s3, s4; variable v1, v2;

Select <s1, a, s4> such that Affects*(a1, a2) and Uses(s1, v1)
and Uses (29, "food")  and Modifies(a, v2)  with s1.stmt#=53
pattern a2("food", _"chili"_) such that Parent(s1, s3) with a1.stmt#=20
```

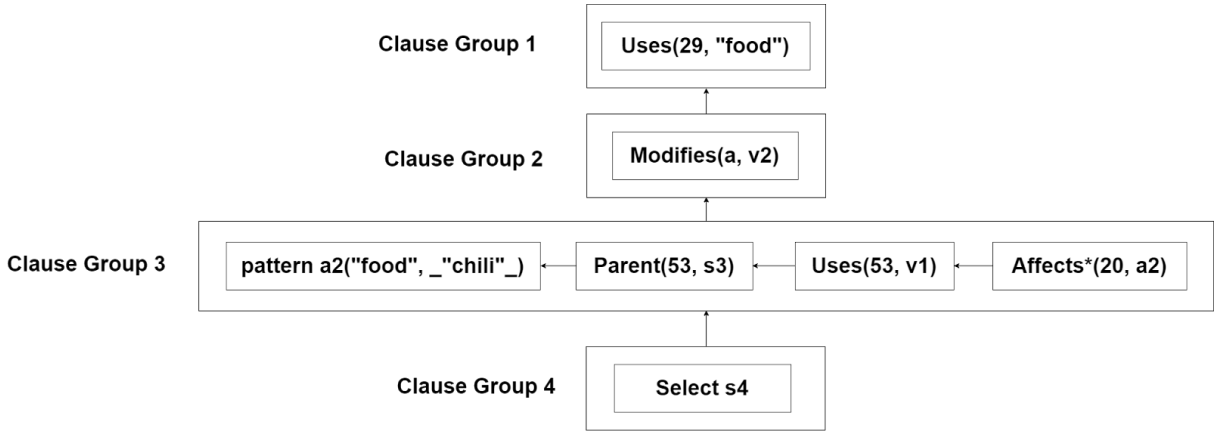
The steps taken by the Optimiser to optimize this query are:

1. The synonym a1 in Affects*(a1, a2) and Next(s3, a1) will be overwritten to 20 due to the last with-clause, turning the Affects* clause to Affects*(20, a2).
2. The costs of each ClauseGroup is computed and stored.
3. Split synonyms into "linked" groups using Union-Find Disjoint Sets (UFDS), by calling union-find on synonyms in the same clause. Here, the groups will be a1, a2, s1, v1, s3, a, v2. Uses(29, "food") will be a lonely clause in a separate group since it's the only clause that has no synonyms. Because the synonym s4 is not present in any of the clauses other than the Select-clause, a new SelectClause will be created that only has the synonym s4. Now, the clauses are split into three Clause groups.



4. Sort clauses within each clause groups based on their costs, then chain them up such that clauses with two synonyms have one of the synonyms evaluated before it is evaluated (unless all clauses in the group has two synonyms).

5. Sort clause groups based on the total cost of clauses in each clause group.



6. Pass the queue of ClauseGroup created to ClauseGroupManager, and pass ClauseGroupManager to the QueryEvaluator.

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 QueryEvaluator will evaluate the clause with the help of appropriate ClauseEvaluators and proceed with the evaluation. These evaluators assist the QueryEvaluator in retrieving the relevant results from the PKB and are produced by providing Clause object to the ResultFactory.

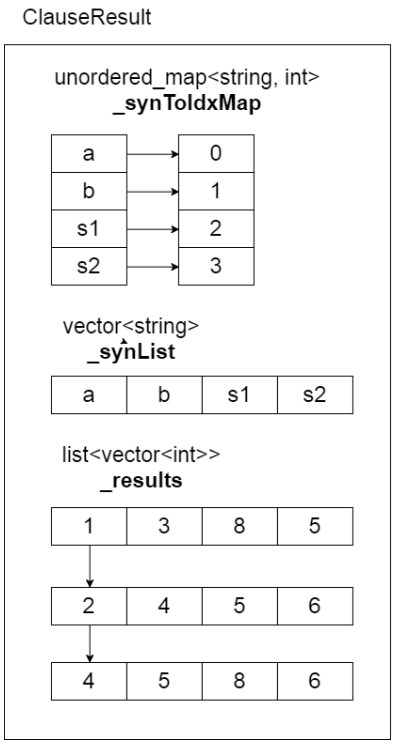


Figure 15: ClauseResult

The ClauseResult is responsible for storing and maintaining the intermediate results 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 satisfy 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 (rows) 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 vectors are used for rows, frequent deletion of columns can be inefficient, since the elements after the deleted column will need to be shifted forward. As such, when doing pruning (more in the Optimiser's section), instead of actually deleting columns, what happens is the column values will be set to -1. When this happens, several previously different rows may now become the same.

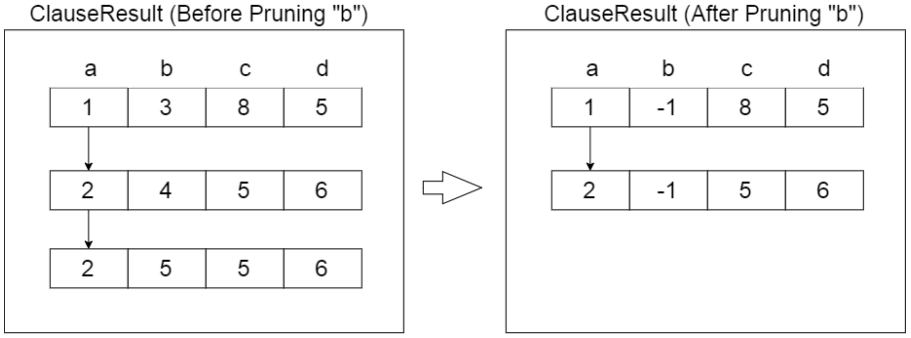


Figure 16: Pruning Synonym "b" in ClauseResult

Duplicates can then be removed and the size of the ClauseResult will be reduced.
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 QueryEvaluator will first initialise the Query-Optimiser, which will split the ClauseObjects based on common synonyms and put them into ClauseGroups. ClauseObjects within each ClauseGroups are then sorted based on the estimated cost and ClauseGroups are also sorted based on the total cost of all the ClauseObjects in them.

2. The sorted ClauseGroups are then passed to the ClauseGroupManager, which is then returned to the QueryEvaluator. The QueryEvaluator can obtain the ClauseGroups, one after the other, from the ClauseGroupManager. The Clause objects will be processed in the groupings and order that the QueryOptimiser has determined.

3. The QueryEvaluator then initialises the ResultFactory and feeds the Clause objects, one at a time, to the ResultFactory that needs to be processed. The ResultFactory will iterate through all the ClauseGroups given by the ClauseGroupManager. Each ClauseGroup will produce a ClauseResult. 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, even if it is from another ClauseGroup.

4. As per Strategy pattern, the ResultFactory will create evaluators specially for the evaluation of the clauses. Figures 17,18 and 19 show the overview of the various evaluators that can be created.

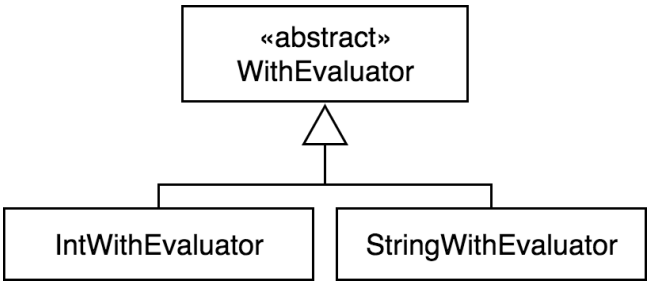


Figure 17: With Evaluator

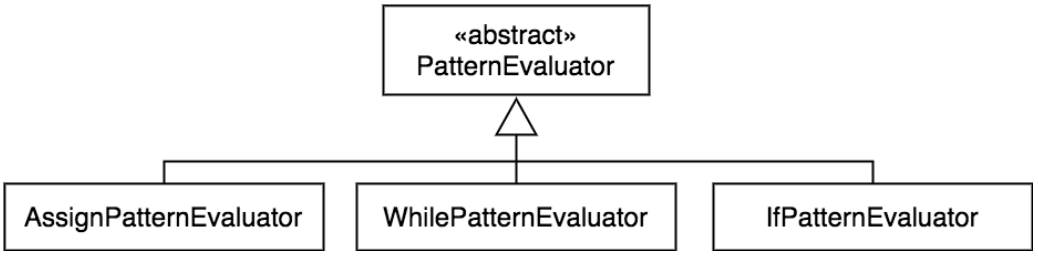


Figure 18: Pattern Evaluator

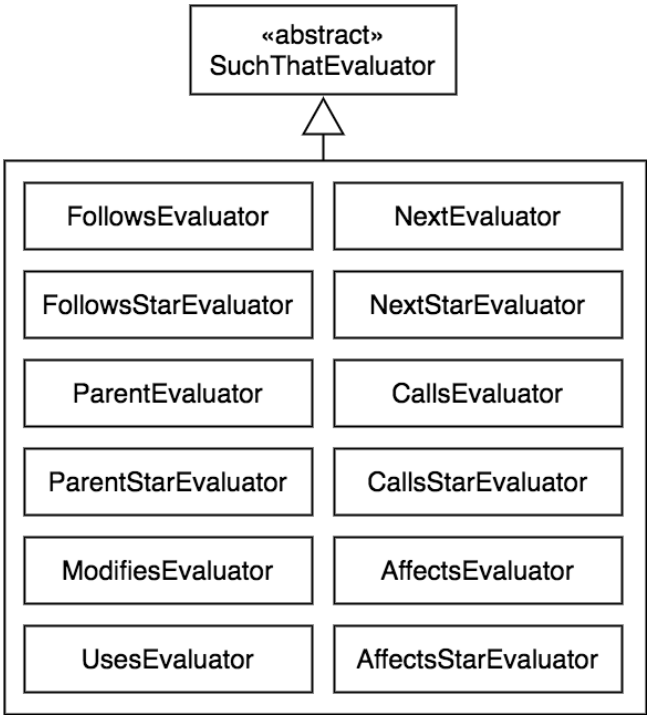


Figure 19: Such That Evaluator

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.

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

6. 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 of the ClauseGroup that the evaluator is currently evaluating. For purely Boolean cases (i.e. Follows(3, 4); Case 1 in Table 4), 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.

7. For clauses that contain 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.

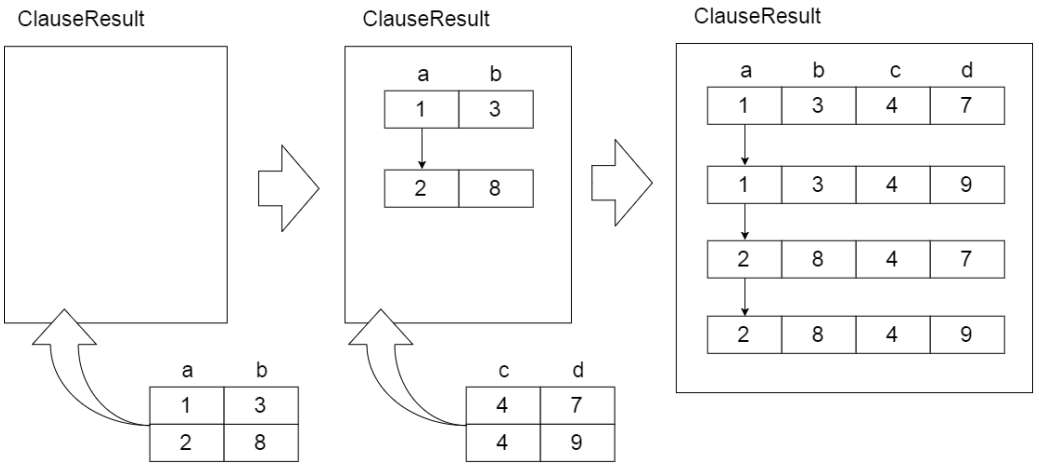


Figure 20: Adding New Single Synonym Results

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

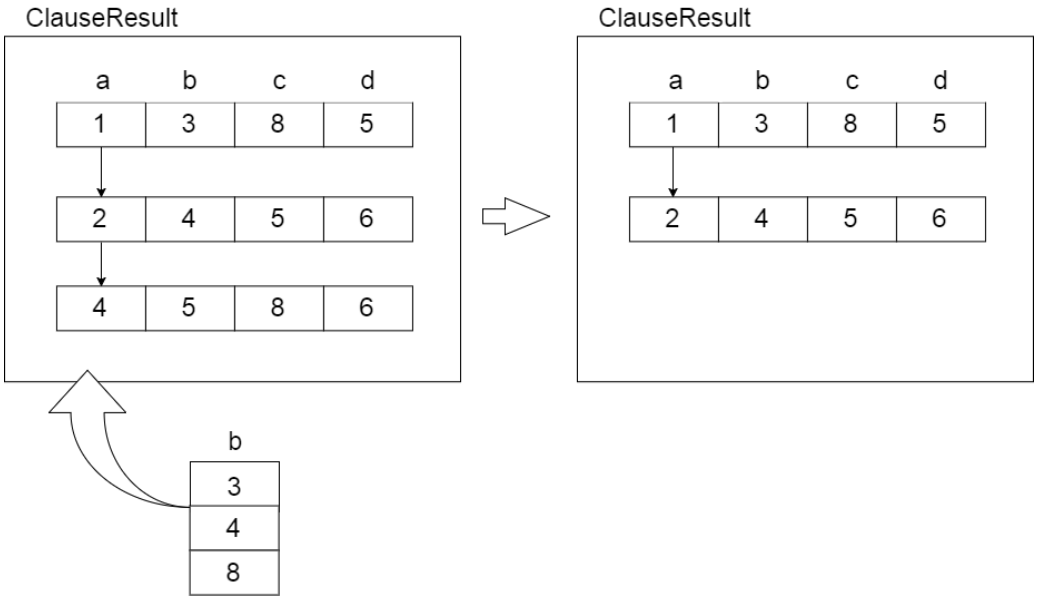


Figure 21: Adding Existing Synonym Results

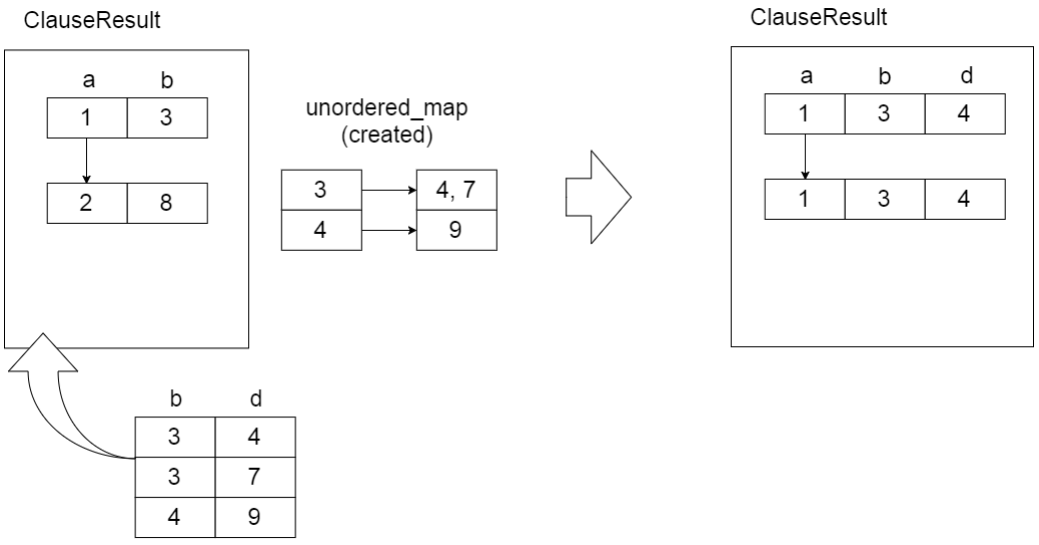


Figure 22: Adding A New Synonym Paired to an Existing Synonym

For the case of 1 new synonym and 1 existing synonym, the ClauseResult will do the merging of result using *hash join*. An unordered_map<int, list<int> > will be created to help with the merging 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 M, the time complexity of this merging process would be $O(M + N)$, which is sufficiently fast.

8. Each ClauseGroup will yield one ClauseResult object. After the ResultFactory has finished processing all the Clause objects in a ClauseGroup, and all of them return true for validity, the factory will pass the evaluated ClauseResult to the ClauseGroupManager. The ClauseGroupManager maintains an internal ClauseResult that only contains all the selected synonyms' result from all ClauseGroups. When the ClauseGroupManager receives a newly evaluated ClauseResult from the QueryEvaluator, it will sieve the results of the selected synonyms and merge them into the internal ClauseResult. After all ClauseGroups has been processed, the ClauseGroupManager will return the final merged ClauseResult of all ClauseGroups to the QueryEvaluator and QueryEvaluator 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,PROCEDURE or CALL.PROCNAME) 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	The Parser has a lot to do while parsing the SIMPLE source code: - Various types of syntax validation - Display error message as user feedback when syntax error is detected - Extract various design abstractions and populate PKB
Alternatives	1. Do all of the job of the parser in one parse 2. 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	1. Time-efficiency 2. Ease of implementation 3. Testability 4. Maintainability 5. Extensibility to future iterations, and possible bonus features
Decision	**Choice: Hybrid** 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 maintain and extend. Therefore, using just a few parses with each parse doing related tasks is 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.

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 is extremely crucial to ensure that data is accessible in the shortest time possible.
Alternatives	- AST - Tables in the form of: • Hash maps • Vector of lists
Criteria for comparison	1. Time complexity of retrieval, addition and deletion 2. Simplicity of implementation 3. Extensibility to future iterations and possible bonus features 4. Testability
Decision	**Choice: Tables in the form of hash maps** Abstract Syntax Tables are the conventional ways to store parsed data. However, when it comes to accessing the data by the Query Evaluator component, it will have 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 an AST allows faster searching, addition, deletion of data. Data structures used in the implementation of tables include: 1. Vectors of lists of strings/integers 2. Hashmap with integer as hash key and list of string/integers as value Amongst the two data structures listed above, hash maps are preferred over vectors due to their fast lookup speeds. To find the desired data in a vector, linear traversal is needed to get to its relative index at a time complexity of $O(n)$, which turns out to be slower than AST, that takes $O(logn)$ time. On top of this, unlike hash maps, vectors does not take strings as index for data mapping. Having multiple hashmaps over ASTs allow faster searching, addition and deletion of data.

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	Hash Maps	Vectors
Data searching	$O(log\ s)$	$O(1)$	$O(s)$
Retrieving single data	$O(log\ s)$	$O(1)$	$O(s)$
Retrieving all the data	$O(s\ log\ s)$	$O(s)$	$O(s)$

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 retrieved 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	- Add another hash maps to store reverse relationships - Self-balancing binary search tree (BST)
Criteria for comparison	1. Time complexity for retrieval 2. Ease of implementation
Decision	**Choice: Add another data structure to store reverse relationships** The time complexity of searching in hash maps is faster compared to searching in trees, as traversing of trees takes $O(logn)$ 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.

3.3.4 PKB Design: CFG Design

Problem	Storage of CFG
Alternatives	- Adjacency List - Adjacency Matrix - Edge List
Criteria for comparison	1. Time complexity for finding relation 2. Time complexity for traversal 3. Ease of implementation
Decision	**Choice: Adjacency List** Time complexity for finding relation: Edge List $O(E)$ > Adjacency List $O(2) = O(1)$ = Adjacency Matrix $O(1)$. Time complexity for traversal: Edge List $O(E)$ > Adjacency Matrix $O(V^2)$ > Adjacency List $O(V)$ 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.

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	- Trees to represent expression - Postfix matching
Criteria for comparison	1. Time complexity of retrieval 2. Simplicity of implementation 3. Testability
Decision	**Choice: Postfix matching** 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.

3.3.6 Affects* Computation Decision

Problem	Computation of any Affects* relation
Alternatives	- Running Affects* algorithm for each query - Computing all Affects* relations and storing it in the cache and then retrieving the required relations
Criteria for comparison	1. Time-efficiency 2. Memory required
Decision	**Choice: Computing all Affect* relations and storing it in the cache then retrieving the required relations** Shorter computation time for future Affects* clauses. As the algorithm for get the boolean result of an Affects* relation such as Affects*(3, 10), it requires the computation of the transitive closure of all Affects relations and checking if it exists. Since the computation time for all Affects relations is about the same as computing all Affects* relations as they are both a one-pass algorithm, it is much faster to compute all Affects* relations and then getting the required relations in $O(1)$ from the cache. Another benefit of doing this is that all the following Affects* relations following the first Affects* clause in the query could be retrieved from the PKB in $O(1)$. For example, Affects*(1, 3) and Affects*(a, 100) and Affects*(a1, a2), the relations can all be retrieved in $O(1)$ following the first Affects*(1, 3) call.

3.3.7 Affects* Design Decision

Problem	Extraction of all Affects* pairs
Alternatives	- Brute force - One-pass
Criteria for comparison	1. Time-efficiency 2. Readability of code
Decision	**Choice: One-pass** Shorter computation time. As Affects* can only be computed on the run during the evaluation of a query, it is very time consuming to run the Affects* algorithm on every assignment statement as it requires the traversal of the CFG n times where n is the number of assignment statements. On top of that, it requires additional time for extra checking of uses and modifies variables in each assignment statement.

3.3.8 Query Validation Design

Problem	1. Difficulty in parsing query 2. Too many if-else string comparisons
Alternatives	1. Usage of Regular Expression 2. Naive tokenizing with specific delimiter 3. Naive if-else
Criteria for comparison	1. Time-efficiency 2. Ease of implementation 3. Testability 4. Maintainability 5. Extensibility to future iterations and possible bonus features
Decision	**Choice: Regular Expression** 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.

Problem	Many "if-else" statements to handle the validation of clauses
Alternatives	Strategy pattern (Handlers and sub-validators)
Criteria for comparison	1. Ease of implementation 2. Ease of extension 3. Testability 4. Memory overhead
Decision	Choice : Strategy Pattern **Justification** 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.

3.3.9 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	1. Strategy pattern (ClauseEvaluator and sub-classes) 2. Factory pattern (ResultFactory and ClauseResult)
Criteria for comparison	1. Ease of implementation 2. Ease of extension 3. Testability 4. Memory overhead
Decision	**Justification: Strategy Pattern** 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. **Justification: Factory Pattern** 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.

3.3.10 Data Structure for Intermediate Result

Problem	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 clause. These operations include merging of tables, insertions, deletions, and many more, which might be computationally expensive if not optimised.
Alternatives	Representation of results: 1. Vector of vectors 2. List of vectors
Criteria for comparison	Time-efficiency is the major criteria, as the process of merging intermediate results is one of the most expensive and most carried-out operation in the query evaluation process.
Decision	**Choice of data structure: Linked-list of vectors** Each vector represents a possible combination of synonym values that satisfy a query, with each index assigned to a particular synonym for $O(1)$ access. Using a linked-list to chain up all the possible combination of synonyms will allow very fast insertion and deletion of existing combination of synonym results in the middle of the list. Insertion/deletion takes $O(1)$. This will enhance performance tremendously,compared to using a vector of vectors.

3.3.11 Query Optimiser - Cost Allocation for Clauses

Problem	How to assign cost to clauses in query
Alternatives	1. Static cost allocation based on statistics and general heuristics 2. Dynamic cost allocation based on information in PKB 3. Hybrid of static and dynamic cost allocation
Criteria for comparison	1. Time-efficiency 2. Ease of implementation 3. Optimization overhead
Decision	**Choice: Static cost allocation based on statistics and general heuristics** Justification 1. The time-efficiency achieved using dynamic cost-allocation is, most of the time, similar to that of static allocation. 2. Ease of implementation 3. Sufficiently efficient for the intended application. Reduce optimisation overhead

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 Abtract 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 LISTSTMT 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. Do refer to Section 5.5 for further details on our System Testing.

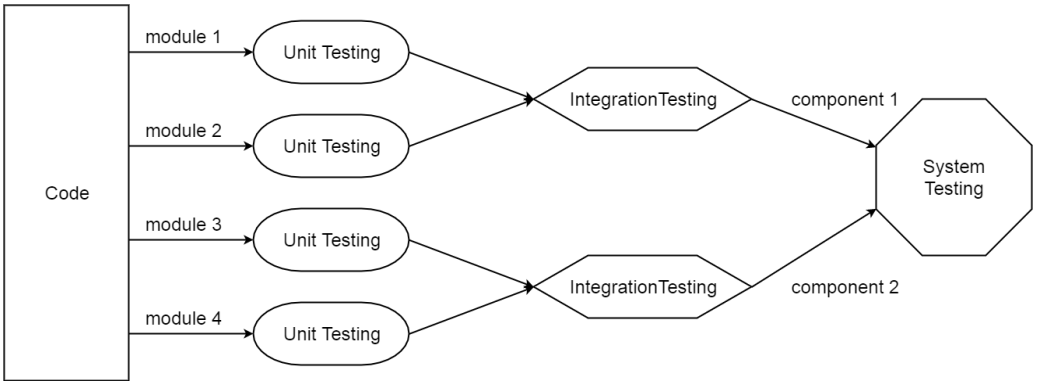
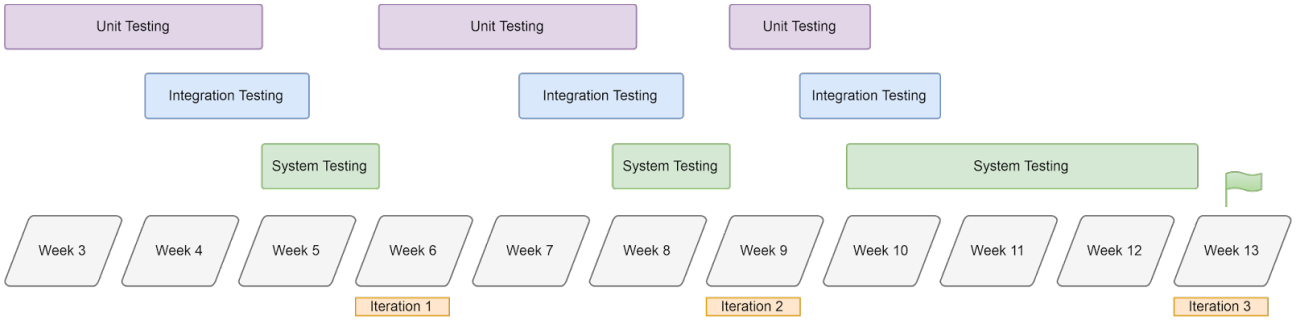


Figure 23: Flow chart showing the testing process

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 the 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

We use 3 different types of procedures to detect bugs within our SPA.

1. Unit testing (white-box testing)
2. Integration testing
3. System testing

The following components are subjected to testing:

Component	Objectives
Parser	• To ensure that the source file is parsed and validated accurately • To ensure that the correct information is stored into PKB
Design Extractor	• To ensure that the computation of transitive closure and computation of star relationships are done correctly
PKB	• To ensure that information can be stored successfully • To ensure that the information retrieved from PKB is accurate and expected
Query Validator	• 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 QueryTree accurately
Query Evaluator	• To ensure that the information retrieved from the QueryTree 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	• To ensure that the information is stored successfully • To ensure that the information is retrieved correctly

5.2 Test Utilities Class

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

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

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.

Test Purpose	To test if PKB computes design entity, stmt, correctly

Required Test Input	`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);`
	`}`

Expected Test Results	true, true, true, true, true, true

Test Purpose	To test if PKB computes relation Calls* correctly
Required Test Input	TEST_METHOD(TestCallsTable) { 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.setModTableProcToVar("Two", "b"); PKB.setModTableProcToVar("Three", "c");

Required Test Input	PKB.setModTableProcToVar("Four", "d");
	PKB.setModTableProcToVar("Five", "e");
	PKB.setModTableProcToVar("Six", "f");
	PKB.setModTableProcToVar("Seven", "g");
	PKB.setModTableProcToVar("Eight", "h");
	PKB.setModTableProcToVar("Nine", "i");
	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)); }

Expected Test Results	true, true, true, true, true, true, true, true, false, false, 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 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	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)); }
Expected Test Results	false, false

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

Test Purpose	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. Cases to consider: 1. Adding new results when ClauseResult is still empty 2. 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	TEST_METHOD(TestAddNewSynPairResults_non EmptyExistingResults_success) { ClauseResult cr; string syn1 = "a"; string syn2 = "b"; list<int> syn2Results{ 4, 5, 6 }; string syn3 = "c"; list<int> syn3Results{ 7, 8, 9 }; cr.updateSynResults(syn1, syn1Results); }

Required Test Input	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
	*********/
	actualResults.sort();
	expectedResults.sort();
	Assert::IsTrue(actualResults == expectedResults); }

Expected Test Results	true

5.3.3 Assertions

We also make use of assertions in our Parser.

```
/*  Removes all result combinations that contains the given value
    for the given synonym.
    Pre-condition: synName must be a synonym that is present in ClauseResult
*/
bool ClauseResult::removeCombinations(string synName, int value)
{    /*
        Note:
        There's no need to remove anything from _synList and _synToIdxMap
        because once a synonym is introduced into the intermetiate result,
        it MUST always have at least 1 possible result.
    */

    assert(ClauseResult::synonymPresent(synName));

    int synIdx = _synToIdxMap.at(synName);
    list<vector<int>>::iterator existingResultsIter = _resultsPtr->begin();
    while (existingResultsIter != _resultsPtr->end())
    {
        if (AbstractWrapper::GlobalStop) {
            return true;   // Consider returning false, check the consequence
        }

        if ((*existingResultsIter).at(synIdx) == value)
            existingResultsIter = _resultsPtr->erase(existingResultsIter);
        else
            existingResultsIter++;
    }
    return 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.

5.4.1 Parser - PKB SubSystem

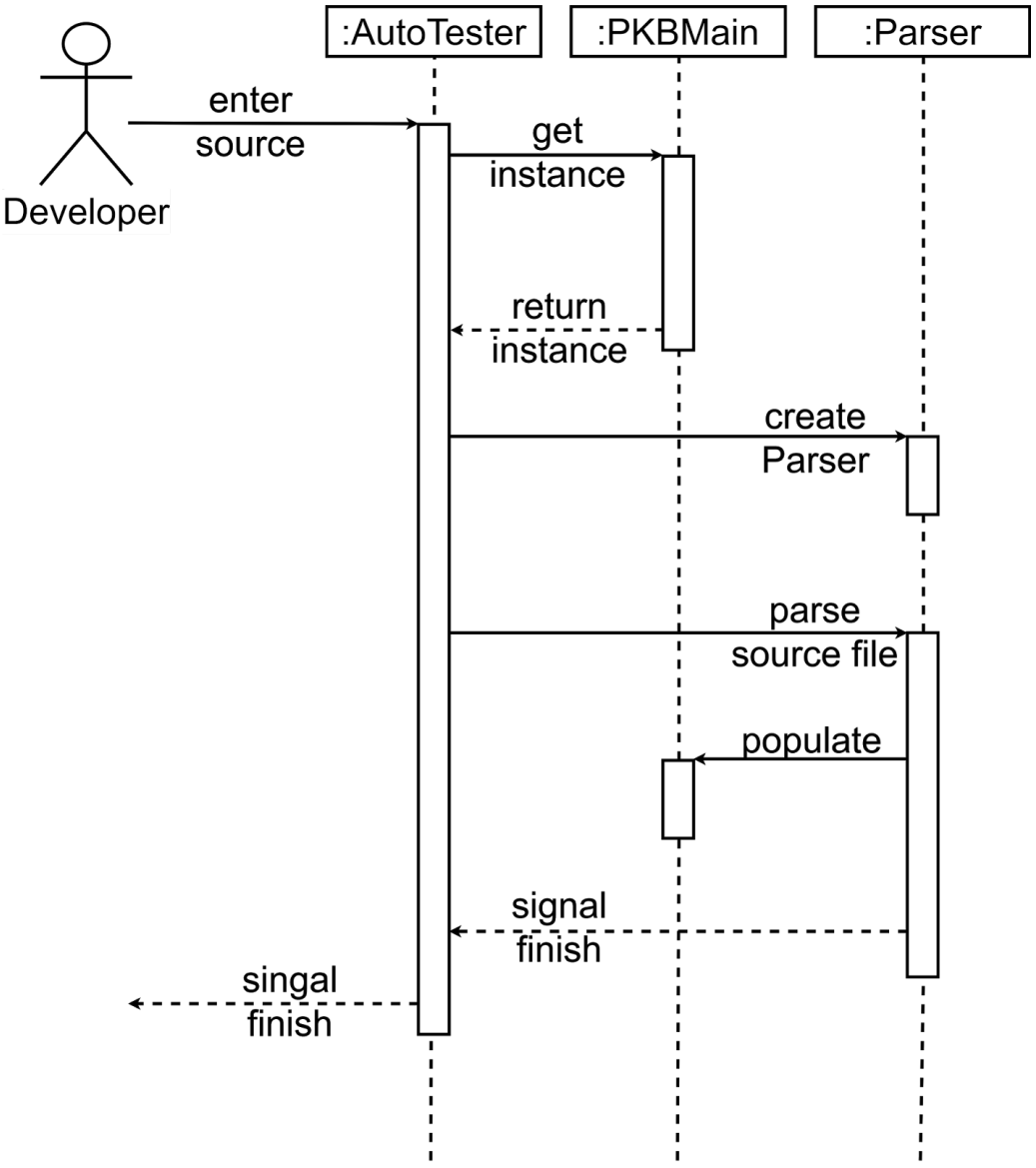


Figure 24: Sequence diagram for Parser-PKB Interaction

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

Test Purpose	To test if PKB stores the relations in the parsed SIMPLE source code correctly.

Required Test Input	TEST_METHOD(TestNextRelation)
	//Set-up
	{ list<int> actualResults;
	list<int> expectedResults;
	Parser parser(pkbPtr);
	Assert::IsTrue(parser.parse
	(dummySimpleSourcePath));//set up previously
	// Simple Next relations
	Assert::IsTrue(pkb.isNext(1, 2));
	Assert::IsTrue(pkb.isNext(2, 3));
	// Next relations when entering if-block
	Assert::IsTrue(pkb.isNext(4, 5));
	Assert::IsTrue(pkb.isNext(16, 17));
	// Next relations when entering
	else-block
	Assert::IsTrue(pkb.isNext(4, 6));
	Assert::IsTrue(pkb.isNext(16, 18)); }

Expected Test Results	true, true, true, true, true, true

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 information 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.

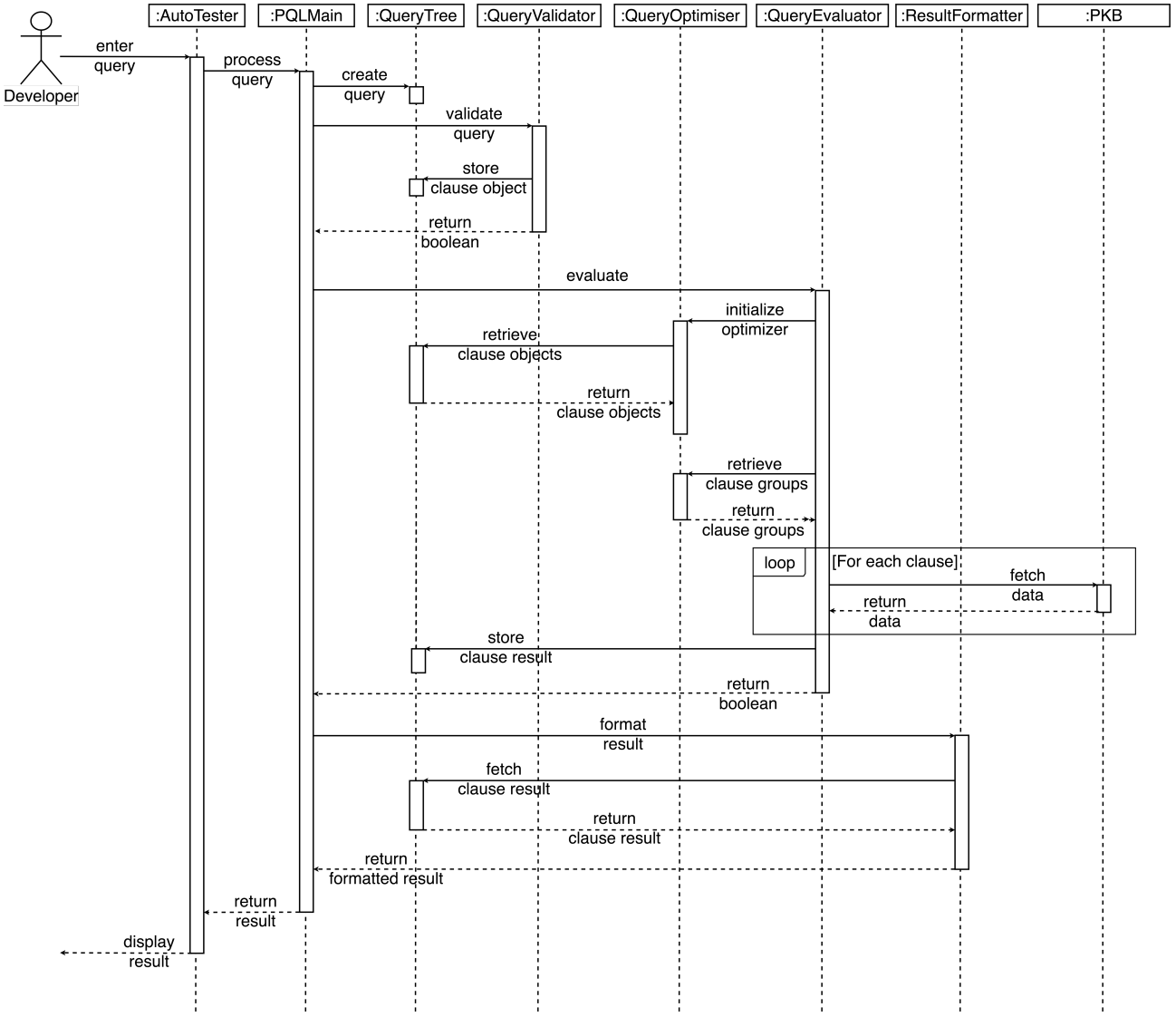


Figure 25: Sequence diagram for PQL-PKB Interaction

Test Purpose	To test if QueryValidator can validate complex query accurately and store the correct data into QueryTree. It specifically targets such that and pattern clauses of different combination.

Required Test Input	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(ÿ,_) such that Follows(s2, s1) and
	Modifies(s1, ẍ) 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(ASSSIGN_PATTERN, "a1", VARIABLE, "v2",
	UNDERSCORE, "_"));
	expectedListPc.push_back(UtilitySelection::
	makePatternClause(WHILE_PATTERN, "w1", IDENT_WITHQUOTES,
	"ÿ", 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)); }

Expected Test Results	true, true, true, true, true, true

5.5 System Tests

System testing involves testing all components of the SPA together. This is carried out with the assistance of the AutoTester. Test cases are designed based on the considerations in Figure 26. Test cases in system testing are controlled and run via a batch file. System test queries are divided into 4 parts, namely:

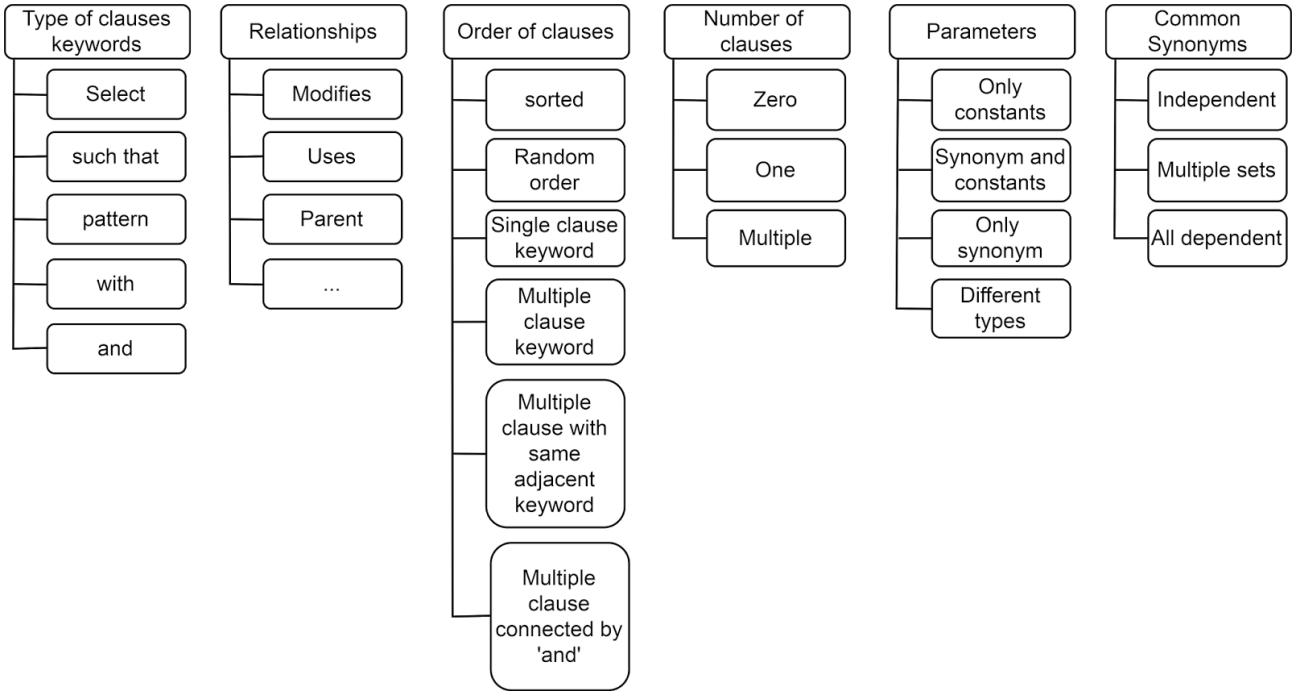


Figure 26: System Testing

1. Simple test queries
2. Complex test queries
3. Validation test queries
4. Stress test queries

Simple test queries are defined as test cases which involve either no clause or only one such that/pattern/with queries. The purpose of these test is to test for each and every relationships and clauses. This can be used to debug PKB component or Query Evaluator component. For example:

```
stmt s;
Select s

stmt s;
Select s with s.stmt# = a.stmt#

assign a;
Select a pattern a(_,_"x"_)

stmt s; variable v;
Select s such that Uses(s,v)

while w1,w2;
Select w2 such that Parent*(w1,w2)
```

They can also be in the form of one such that clause and one pattern/with clause, for example:

```
while w1,w2;
```

```
Select w2 such that Parent*(w1,w2) pattern w1(v,_)

stmt s; call c;
Select c such that Next*(a,c) with a.stmt# = 12
```

Complex test queries are defined as test cases that involves multiple clauses, common synonym or all clauses, etc. The purpose of the complex queries not only test whether the queries are evaluated correctly, but also used to ensure optimiser is working properly so as to reduce Cartesian products.

For example:

```
while w1,w2; call c;
Select w2 such that Parent*(w1,w2) and Next(w1,c)

stmt s; assign a; variable v; procedure p;
Select <a,v> such that Uses(a,v) and Uses(s,v) with
v.varName = p.procName

while w1,w2; assign a1,a2; if ifs; constant c;
Select <w1,w2,a1,a2> such that Affects*(a1,a2) such that Next(w1,ifs)
with w1.stmt# = c.value such that Parent*(w2,a1) pattern a1(_,_"osaka"_)
```

Validation test queries are defined as test cases that check the validity of the queries. The following queries are considered invalid due to error in syntax or declaration and thus returns 'none'. The test is used in debugging the Query Validator component.

```
stmt s;
Select a

stmt s; variable v;
Select s such that Uses(s,v);

assign a;;;;
Select a

stmt s; stmt s;
Select s

stmt s;
Select s.varName
```

The following system test queries check for whitespaces and they are considered valid and should return results accordingly.

```
stmt s;
Select     s      with     s.stmt     =     12

stmt s;
Select <   s   ,   s   >
```

```
stmt s;
       Select      s.stmt
```

Stress test queries are queries that tests how many queries are computed on the fly (Next*, Affects and Affects*) and how many tuples our SPA can handle.The purpose of the stress queries is to evaluate the performance and effectiveness of the program under a heavy load. Example of such queries are shown:

```
stmt s1,s2,s3;
Select s1 such that Next*(s1,s2) and Next*(s2,s3)

stmt s1,s2,s3,s4;
Select s1 such that Next*(s1,s2) and Next*(s2,s3) and Next*(s3,s4)

stmt s1,s2,s3;
Select s1 such that Affects*(s1,s2) and Affects*(s2,s3)

stmt s1,s2,s3,s4;
Select s1 such that Affects*(s1,s2) and Affects*(s2,s3)
and Affects*(s3,s4)

stmt s1,s2,s3;
Select <s1,s2> such that Affects*(s1,s2) and Affects*(s2,s3)

stmt s1,s2,s3;
Select <s1,s2,s3> such that Affects*(s1,s2) and Affects*(s2,s3)
```

5.5.1 Testing Methods

The team has grouped similar test cases into the same folder. Each folder clearly describes its purpose, as shown in the table below. Do note that internally, we refer to our SPA as SPAXI.

Item	Description
SampleTest	Contains sample tests provided at the start of the project
SimpleTest	Contains 3 SIMPLE source and query files
FocusTest	Contains 3 sub-folders • Sequential (Targets Follows/*, pattern assign, with) • Loop (Targets Parent/*, Modifies/Uses. Affects/* etc.) • InterProcedural (Targets Calls/*, Next/*, etc.)
StressTest	Contains long SIMPLE source • Targets Next/* and Affects/* • Targets combination of Clauses that amounts to large intermediate result, but improves performance with optimisation techniques in place
ValidationTest	Provides a second-layer check and final confirmation on validation after incorporating the different components of the system
TestResult	Stores all xml result generated by running tests Contains a sub-folder • cmd: Stores output from command prompt
AutoTester.exe	Executable that evaluates queries based on the given source and outputs the result
SpaXI.bat	A batch file that provides an interactive command prompt interface

Figure 27: Organization of Folders for System Tests

As there are many query files to be written, the team came up with a way to automate testing. Firstly, for each of the folders above, an Excel spreadsheet is created to hold all the queries, as shown in image Excel. In the Excel, there may contain more than one sheet. By using Excel, many repetitive work can be omitted. The autocomplete feature of Excel also helps in automatic numbering of index, as well as generating some of the queries. Secondly, a python script was written to assist the group in generating the query text and converting them into text files shown in image query . Lastly, the team created a command prompt interface, called SpaXI, to facilitate the rest of testing. After which, results are stored in TestResult folder.

```
1 - All statements HaveResult
stmt s;
Select s
1,2,3,4,5,6,7,8
5000
2 - All assignments HaveResult
assign a;
Select a
1,2,4,5,6,7,8
5000
3 - All whiles HaveResult
while w;
Select w;
3
5000
4 - All variables HaveResult
variable v;
Select v;
a,b,c,d,i,j
5000
```

Figure 28: Standard Query Format

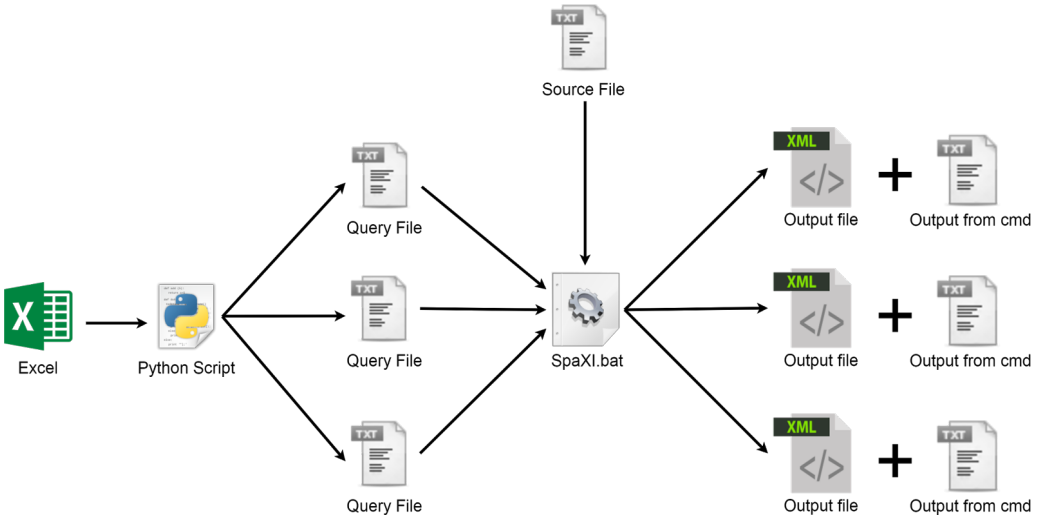


Figure 29: General flow of Automatic Testing

Introducing SpaXI SpaXI.bat, a batch file is created to provide an interactive command prompt interface for testing. The interface is shown below.

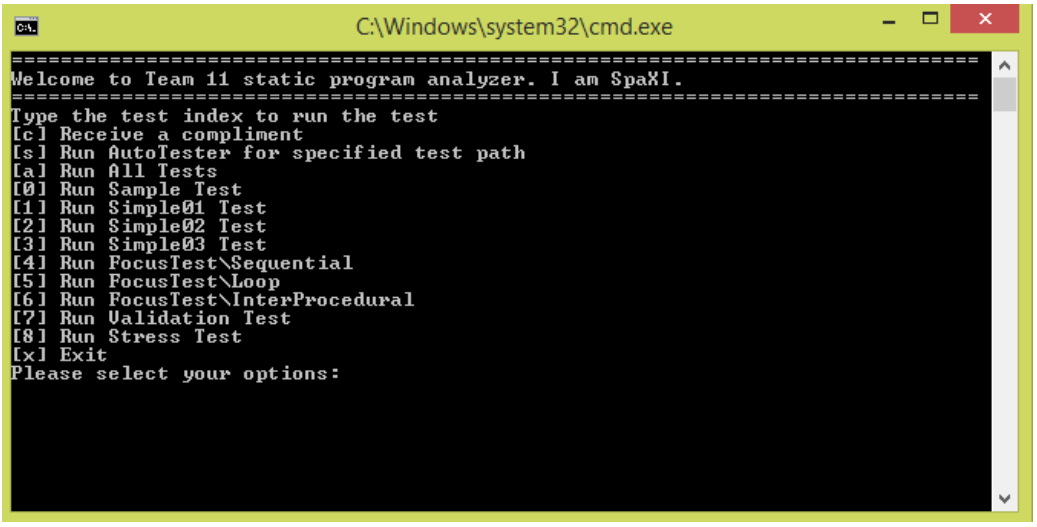


Figure 30: SPAXI's Interface

Command [c] : To receive a compliment from SpaXi to brighten up your day
Command [s]: To run an external source and query file provided by user
Command [a]: To run all tests. At the end of the test, a summary text file will be generated, as shown in Figure 29. A tag called 'BUG' and/or ;TIMEOUT' will be attached if the test fails or gets timed out.
Command [?]: Enter the index of the test folder to run all the tests in that folder. ('?') represents an integer index. For example, if '3' is entered as a command, SpaXi will run Simple03 Test.)
Command [x] : Exit SpaXi

```
Overall Test Ran    : 244
Overall Test Passed : 235
Overall Test Failed : 6
Overall Test Timeout: 3


TestResult\1_SampleTest.xml:
Test Ran: 25    [BUG][TIMEOUT]
Passed   : 22
Failed   : 1
Timeout  : 2
```

Figure 31: Summary of all tests run by SPAXI

5.6 Sample Queries

Test Purpose	To test if simple/complex queries produce the expected results 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 Parent IntSynonym BehindWhile HaveResult stmt s; Select s such that Parent(3, s) 8 5000 3 - Select SingleSynonym Assign FollowsStar UnderscoreInt HaveResult assign a; Select a such that Follows*(a, 3) 1,2,3,4,5,6,7,8 5000 4 - Select SingleSynonym Var Uses UnderscoreInt NoResult variable v; Select v such that Uses(v, "x") a, b, c 5000 5 - Select SingleSynonym Modifies UnderscoreUnderscore Invalid variable v; Select v such that Modifies(_, v) none 5000

Required Test Input	6 - Select SingleSynonym Call UnderscoreSynonym HaveResult call c; Select c.stmt# such that Follows*(1, c) 2,3 5000 7 - Select SingleSynonym Call Uses Synonym HaveResult call c; procedure p; Select c.procName such that Uses(p, "x") with c.procName = p.procName 1 5000 8 - Select SingleSynonym Stmt FollowsStar SynonymInt NoResult stmt s; Select s such that Follows*(s, 1) none 5000 9 - Select SingleSynonym Stmt Parent SynonymInt BeforeWhile HaveResult stmt s; Select s such that Parent(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

Test Purpose	To test performance for complex queries, query optimiser, Affects and Affects*
Required Test Input	1 - Select SingleSynonym Stmt SuchThat Follows IntInt and Follows IntInt HaveResult stmt s; assign a; Select s such that Follows(1, s) and Affects(a, 10) with s.stmt# = a.stmt# 5, 6 5000 2 - 1 - 2 cmn syn + 1 cmn syn + 1 cmn syn assign a1,a2,a3,a4; stmt s; call cl; if ifs; while w; procedure p; variable v; Select ifs such that Next*(a1,a2) and Affects(a2,a3) with a3.stmt#=s.stmt# such that Parent(w,a4) pattern a4(_,_"tokyo"_) such that Follows*(ifs,cl) and Uses(cl,v) with v.varName = p.procName 370,376,418 5000 3 - 1 - 3 Affects assign a1,a2,a3,a4; Select a3 such that Affects(a1, a2) and Affects(a2, a3) and Affects(a3, a4)# 3, 7, 17, 30, 32, 46, 47, 54, 56, 57, 59, 60, 61, 62, 63, 65, 67, 68, 83, 85, 86, 87, 89, 91, 92, 95, 96, 97, 99, 108, 113, 116, 117, 118, 121, 122, 124, 125, 127, 128, 131, 143, 144, 145, 147, 151, 153, 154, 156, 158 5000 3 - 2 - 4Affects assign a1,a2,a3,a4,a5; Select a4 such that Affects(a1, a2) and Affects(a2, a3) and Affects(a3, a4) and Affects(a4,a5) 7, 32, 46, 47, 54, 56, 57, 59, 60, 62, 63, 65, 67, 68, 83, 85, 86, 87, 89, 91, 92, 95, 96, 97, 99, 113, 116, 117, 118, 121, 122, 124, 125, 127, 128, 131, 143, 144, 145, 153, 154, 156, 158 5000 4 - 4 - 6Affects assign a1,a2,a3,a4,a5,a6,a7; Select a6 such that Affects(a1, a2) and Affects(a2, a3) and Affects(a3, a4) and Affects(a4,a5) and Affects(a5, a6) and Affects(a6, a7) 46, 47, 54, 56, 57, 59, 60, 62, 63, 65, 67, 68, 83, 85, 86, 87, 89, 91, 92, 95, 96, 97, 99, 113, 116, 117, 118, 121, 122, 124, 125, 127, 128, 131, 143, 144, 145, 153, 154, 156, 158 5000 5 - Select SingleSynonym Stmt SuchThat Follows IntInt SuchThat Follows IntInt HaveResult stmt s; assign a; var v; call c; Select a such that Parent*(1, a) and Follows(2, c) pattern a(v,_"x"_) with v.varName = c.procName 2, 8 5000

5.7 Testing Statistics

5.7.1 Unit Testing and Integration Testing

We collectively wrote 983 unit tests and integration tests.

5.7.2 System Testing

1. Source Files (30-500 lines) : 9
2. Number of queries simple queries : 2009
3. Number of complex queries (including validation and stress tests) : 860

6 Discussion

6.1 Technical Difficulties

For iteration 3, the technical difficulties were minimal as the team was very familiar with Visual Studio 2015, C++ and Version Control.

6.2 Project Management

We discussed what could be improved following Iteration 2 and how we could further enhance our project management skills. We came up with mini-iterations for every week, and scheduled weekly meetings in order to ensure optimal working conditions and motivation for each team member to put their best effort in. One team member was put in charge of scheduling and made sure to keep everyone on track. We also updated each other on any changes to be made to the schedule, but by and large we stuck to what we planned for this Iteration, and we can definitely say that our project management skills have improved. This way, we were also able to input consistent work.

6.3 Workload and Time Management

Two challenges we faced were trying to design an efficient algorithm to compute star relationships for Affects*, and how to system test effectively. However, with effective time management and divisions into mini-iterations, our team stuck to the deadline, and each member managed to complete these challenging tasks.

A Documentation of Abstract APIs

Given on the next page.

PKBMain API for Parser

BOOLEAN addVariable(STRING var) If the variable is not found in the Variable Index Table, add the variable to the table and returns true.
BOOLEAN addProcedure(STRING proc) If the procedure is not found in the Procedure Index Table, add the procedure and returns true.
BOOLEAN addVariable(STRING var) If the variable is not found in the Variable Index Table, add the variable and returns true. Otherwise, return false.
BOOLEAN addAssignmentStmt(INTEGER stmt) If the assignment statement is not found in the Statement Type List, add the statement and returns true.
BOOLEAN addWhileStmt(INTEGER stmt) If the while statement is not found in the Statement Type List, add the statement and returns true.
BOOLEAN addWhileStmt(INTEGER stmt, STRING controlVar) If the while statement and its control variable is not found in the varToWhileMap inside Pattern Table, add them and return true.
BOOLEAN addIfStmt(INTEGER stmt) If the if-else statement is not found in the Statement Type List, add the statement and returns true.
BOOLEAN addIfStmt(INTEGER stmt, STRING controlVar) If the if-else statement and its control variable is not found in the varToIfMap inside Pattern Table, add them and return true.
BOOLEAN addConstant(INTEGER stmt, INTEGER constant) Returns true if constant is added successfully.
BOOLEAN setFollowsRel (INTEGER stmtBef, INTEGER stmtAft) Returns true if Follows(stmtBef, stmtAft) is added successfully to the Follows Table.
BOOLEAN setParentChildRel (INTEGER parentStmt, INTEGER childStmt) Returns true if Parent(parentStmt, childStmt) is added successfully to the Parent Table.

BOOLEAN setCallsRel (INTEGER stmt, STRING callerProcName, STRING calleeProcName) Returns true if Calls(callerProcName, calleeProcName) is added successfully to the Calls Table
BOOLEAN setModTableStmtToVar(INTEGER stmt, STRING var) Returns true if Modifies(stmt, var) is added successfully to the Modifies Table.
BOOLEAN setModTableProcToVar(STRING proc, STRING var) Returns true if Modifies(proc, var) is added successfully to the Modifies Table.
BOOLEAN setUsesTableStmtToVar(INTEGER stmt, STRING var) Returns true if Uses(stmt, var) is added successfully to the Uses Table.
BOOLEAN setUsesTableProcToVar(STRING proc, STRING var) Returns true if Uses(proc, var) is added successfully to the Uses Table.
BOOLEAN setPatternRelation(INTEGER stmt, STRING var, STRING expression) Returns true if the pattern relation is added successfully to the Pattern Table

PKBMain API for QueryEvaluator

General purpose API
LIST<INTEGER> getAllProcedures() Returns all the procedures stored in PKB.
LIST<INTEGER> getAllStatements() Returns all the statements stored in PKB.
LIST<INTEGER> getAllVariables() Returns all the variables stored in PKB.
LIST<INTEGER> getAllConstants() Returns all the constants stored in PKB.
LIST<INTEGER> getAllAssignments()

Returns all the assignment statements stored in PKB.
LIST<INTEGER> getAllWhiles() Returns all the while statements stored in PKB.
LIST<INTEGER> getAllIfs() Returns all the if-else statements stored in PKB
LIST<INTEGER> getAllCallsStmt() Returns all the call statements stored in PKB
LIST<STRING> getAllVarNames() Returns a list of strings of variable names.
LIST<STRING> getAllProcNames() Returns a list of strings of procedure names.
BOOLEAN isPresent(INTEGER stmtNum) Returns true if stmtNum exists in PKB.
BOOLEAN isPresent(STRING varName) Returns true if varName exists in PKB.
BOOLEAN isProcedure(STRING procName) Returns true if procName is a procedure.
BOOLEAN isVariable(STRING varName) Returns true if varName is a variable.
BOOLEAN isAssignment(INTEGER stmtNum) Returns true if stmtNum is an assignment statement.
BOOLEAN isWhile(INTEGER stmtNum) Returns true if stmtNum is an while statement.
BOOLEAN isIf(INTEGER stmtNum) Returns true if stmtNum is an if-else statement.

BOOLEAN isCall(INTEGER stmtNum) Returns true if stmtNum is an call statement.
BOOLEAN isConstant(INTEGER c) Returns true if there exists constant c.
BOOLEAN isProgLine(INTEGER progLine) Returns true if there exists progLine.
Utility API
BOOLEAN isInstanceOf(ENTITY type, INTEGER arg) **BOOLEAN isInstanceOf(ENTITY type, STRING arg)** Returns true if arg.type == type.
LIST<INTEGER> convertStringToIdx(LIST<STRING> stringList, ENTITY type) Returns a list of integer indexes of a desired statement type from a list of strings.
LIST<STRING> convertIdxToString(LIST<INTEGER> indexList, ENTITY type) Returns a list of string of names from a list of integer statement s, such that s.type == type.
LIST<INTEGER> getAllOfInt(ENTITY type) Returns a list of integer statement s, such that s.type == type.
LIST<STRING> getAllStringOf(ENTITY type) Returns a list of string of variable names, callee names or procedure names depending on the entity type passed in the type parameter
BOOLEAN isSameName(ENTITY type1, INTEGER lhs, ENTITY type2, INTEGER rhs) Returns true if both lhs and rhs indexes has mapped to the same string.
Follows API
BOOLEAN isFollows(INTEGER s1, INTEGER s2) Returns true if Follows(s1,s2) holds.
BOOLEAN isBefore(INTEGER s) Returns true if Follows(s, _) holds.

BOOLEAN isAfter(INTEGER s) Returns true if Follows(_,s) holds.
BOOLEAN hasFollows() Returns true if Follows(_,_) holds.
INTEGER getBefore(INTEGER s) Returns an integer statement s1 such that Follows(s1, s) holds. Otherwise return 0.
INTEGER getAfter(INTEGER s) Returns an integer statement s1 such that Follows(s, s1) holds. Otherwise return 0.
LIST<INTEGER> getAllBefore(ENTITY type) Returns a list of statement numbers if there exists a statement s1 such that Follows(s1,_) and s1.type == type. Otherwise returns an empty list.
LIST<INTEGER> getAllAfter(ENTITY type) Returns a list of statement numbers if there exists a statement s1 such that Follows(_,s1) and s1.type == type. Otherwise returns an empty list.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllFollows(ENTITY type1, ENTITY type2) Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both integer lists, Follows(s1,s2) holds.
Follows* API
BOOLEAN isFollowsStar(INTEGER s1, INTEGER s2) Returns true if Follows*(s1,s2) holds.
BOOLEAN isBefore(INTEGER s) Returns true if Follows*(s, _) holds.
BOOLEAN isAfter(INTEGER s) Returns true if Follows*(_,s) holds.
BOOLEAN hasFollows() Returns true if Follows*(_,_) holds.

INTEGER getBeforeStar(INTEGER s) Returns an integer statement s1 such that Follows*(s1, s) holds. Otherwise return 0.
INTEGER getAfterStar(INTEGER s) Returns an integer statement s1 such that Follows*(s, s1) holds. Otherwise return 0.
LIST<INTEGER> getAllBeforeStar(ENTITY type) Returns a list of statement numbers if there exists a statement s1 such that Follows*(s1,_) and s1.type == type. Otherwise returns an empty list.
LIST<INTEGER> getAllAfterStar(ENTITY type) Returns a list of statement numbers if there exists a statement s1 such that Follows*(_,s1) and s1.type == type. Otherwise returns an empty list.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllFollows(ENTITY type1, ENTITY type2) Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both integer lists, Follows*(s1,s2) holds.
Parents API
BOOLEAN isParentChild(INTEGER s1, INTEGER s2) Returns true if Parent(s1,s2) holds.
BOOLEAN isParent(INTEGER s) Returns true if Parent(s,_) holds.
BOOLEAN isChild(INTEGER s) Returns true if Parent(_,s) holds.
BOOLEAN hasParentRel() Returns true if Parent(_,_) holds
LIST<INTEGER> getParent(INTEGER s, ENTITY type) Returns a list of integer statement s1 such that Parent(s1, s) holds and s1.type == type.
LIST<INTEGER> getChildren(INTEGER s, ENTITY type) Returns a list of integer statement s1 such that Parent(s, s1) holds and s1.type == type.

LIST<INTEGER> getAllParents(ENTITY type) Returns a list of integer statement s1 such that Parent(s1, _) holds and s1.type == type.
LIST<INTEGER> getAllChildren(ENTITY type) Returns a list of integer statement s1 such that Parent(_, s1) holds and s1.type == type.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllParentsRel() Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both integer lists, Parent(s1,s2) holds.
Parents* API
BOOLEAN isParentStarChild(INTEGER s1, INTEGER s2) Returns true if Parent*(s1,s2) holds.
BOOLEAN isParent(INTEGER s) Returns true if Parent*(s,_) holds.
BOOLEAN isChild(INTEGER s) Returns true if Parent*(_,s) holds.
BOOLEAN hasParentRel() Returns true if Parent*(_,_) holds
LIST<INTEGER> getParentStar(INTEGER s, ENTITY type) Returns a list of integer statement s1 such that Parent*(s1, s) holds and s1.type == type.
LIST<INTEGER> getChildrenStar(INTEGER s, ENTITY type) Returns a list of integer statement s1 such that Parent*(s, s1) holds and s1.type == type.
LIST<INTEGER> getAllParents(ENTITY type) Returns a list of integer statement s1 such that Parent*(s1, _) holds and s1.type == type.
LIST<INTEGER> getAllChildren(ENTITY type) Returns a list of integer statement s1 such that Parent*(_, s1) holds and s1.type == type.

PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllParentStarRel() Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both integer lists, Parent*(s1,s2) holds.
Uses API
BOOLEAN isUses(INTEGER s, INTEGER v) Returns true if Uses(s, v) holds.
BOOLEAN isUsesProc(INTEGER p, INTEGER v) Returns true if Uses(p, v) holds
BOOLEAN isUsingAnything(INTEGER s) Returns true if Uses(s, _) holds.
BOOLEAN isUsingAnythingProc(INTEGER p) Returns true if Uses(p, _) holds.
LIST<INTEGER> getUsesFromStmt(INTEGER s) Returns a list of variables v in the form of varIdx, such that Uses(s, v) holds.
LIST<INTEGER> getUsesFromProc(INTEGER p) Returns a list of variables v in the form of varIdx, such that Uses(p, v) holds.
LIST<INTEGER> getUsesFromVar(STRING v, ENTITY type) Returns a list of statement number s such that Uses(s, v) holds and s.type == type.
LIST<INTEGER> getProcUsesFromVar(STRING v) Returns a list of procedure indexes such that Uses(p, v) holds.
LIST<INTEGER> getStmtThatUsesAnything(ENTITY type) Returns a list of statement number s such that Uses(s, _) holds and s.type == type.
LIST<INTEGER> getProcThatUsesAnything() Returns a list of procedure indexes p such that Uses(p, _) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getUsesPairs(ENTITY type)

Returns a pair of integer lists containing statement s (where s.type == type) and variable v in the form of varIdx, such that in the same index of both lists, Uses(s, v) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getProcUsesPairs(ENTITY type) Returns a pair of integer lists containing procedure indexes p and variable v in the form of varIdx, such that in the same index of both lists, Uses(p, v) holds.
Modifies API
BOOLEAN isMod(INTEGER s, INTEGER v) Returns true if Modifies(s, v) holds.
BOOLEAN isModProc(INTEGER p, INTEGER v) Returns true if Modifies(p, v) holds
BOOLEAN isModifyingAnything(INTEGER s) Returns true if Modifies(s, _) holds.
LIST<INTEGER> getModifiesFromStmt(INTEGER s) Returns a list of variables v in the form of varIdx, such that Modifies(s, v) holds.
LIST<INTEGER> getModifiesFromProc(INTEGER p) Returns a list of variables v in the form of varIdx, such that Modifies(p, v) holds.
LIST<INTEGER> getModifiesFromVar(STRING v, ENTITY type) Returns a list of statement number s such that Modifies(s, v) holds and s.type == type.
LIST<INTEGER> getProcModsFromVar(STRING v) Returns a list of procedure indexes such that Modifies(p, v) holds.
LIST<INTEGER> getStmtThatModifiesAnything(ENTITY type) Returns a list of statement number s such that Modifies(s, _) holds and s.type == type.
LIST<INTEGER> getProcThatModifiesAnything() Returns a list of procedure indexes p such that Modifies(p, _) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getModifiesPairs(ENTITY type) Returns a pair of integer lists containing statement s (where s.type == type) and variable v in the form

of varIdx, such that in the same index of both lists, Modifies(s, v) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getProcModifiesPairs(ENTITY type) Returns a pair of integer lists containing procedure indexes p and variable v in the form of varIdx, such that in the same index of both lists, Modifies(p, v) holds.
Pattern API
LIST<INTEGER> getAllAssignments() Returns a list of assignment statement numbers, when the query evaluator calls pattern a(_,_)
LIST<INTEGER> getAllAssignments(INTEGER v) Returns a list of assignment statement numbers, such that pattern a(v,_) holds.
BOOLEAN isPartialMatch(INTEGER a, INTEGER v, STRING expression) Returns true if assignment statement a modifies v, and has partial match with expression being inputted as a parameter partially match with the right hand side expression of the corresponding statements.
BOOLEAN isExactMatch(INTEGER a, INTEGER v, STRING expression) Returns true if assignment statement a modifies v, and has partial match with expression being inputted as a parameter exactly match with the right hand side expression of the corresponding statements.
LIST<INTEGER> getPartialMatchStmt(STRING expression) Returns a list of assignment statement numbers, where the expression being inputted as a parameter partially match with the right hand side expression of the corresponding statements. i.e. a(_,_expression_) holds.
LIST<INTEGER> getExactMatchStmt(STRING expression) Returns a list of assignment statement numbers, where the expression being inputted as a parameter exactly match with the right hand side expression of the corresponding statements. i.e. a(_,expression) holds.
LIST<INTEGER> getPartialBothMatches(INTEGER v, STRING expression) Returns a list of assignment statement numbers, where the variable index v corresponds to its left hand side expression, and the expression being inputted as a parameter partially match with the right hand side expression of its corresponding statements. i.e. a(v, _expression_) holds.
LIST<INTEGER> getExactBothMatches(INTEGER v, STRING expression) Returns a list of assignment statement numbers, where the variable index v corresponds to its left hand

side expression, and the expression being inputted as a parameter exactly match with the right hand side expression of its corresponding statements. i.e. a(v, expression) holds.
LIST<INTEGER> getPartialMatchVar(INTEGER a, STRING expression) Returns a list of left hand side variable indexes v, where the expression being inputted as a parameter partiallly match with the right hand side expression of statement a. i.e. a(v, expression) holds.
LIST<INTEGER> getExactMatchVar(INTEGER a, STRING expression) Returns a list of left hand side variable indexes v, where the expression being inputted as a parameter partiallly match with the right hand side expression of statement a. i.e. a(v, expression) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getLeftVariables() Returns a pair of integer lists, where the first list corresponds to the statement number s, and the second list corresponds to the variable indexes v, such that in the same index of both list, pattern a(v_) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getLeftVariablesThatPartialMatchWith(STRING expression) Returns a pair of integer lists, where the first list corresponds to the statement number s, and the second list corresponds to the variable indexes v, such that in the same index of both list, pattern a(v_expression_) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getLeftVariablesThatExactMatchWith(STRING expression) Returns a pair of integer lists, where the first list corresponds to the statement number s, and the second list corresponds to the variable indexes v, such that in the same index of both lists, pattern a(v,expression) holds.
LIST<INTEGER> getAllWhiles() Returns a list of while statement numbers, when query evaluator calls w(_,_)
LIST<INTEGER> getWhilesWithControlVariable(INTEGER v) Returns a list of while statement numbers with the control variable v.
BOOLEAN isWhileControlVar(INTEGER w, INTEGER v) Returns true if the while statement w has the control variable v.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getControlVariablesInWhile() Returns a pair of integer lists, where the first list corresponds to the while statement number, and the second list corresponds to the variable indexes v, such that in the same index of both lists, pattern

w(v,_) holds.
LIST<INTEGER> getControlVariablesInWhile(INTEGER w) Returns a list of control variables in the form of varIdx which corresponds to the while statement w.
LIST<INTEGER> getWhileFromControlVar(INTEGER v) Returns a list of while statements that has a control variable v
LIST<INTEGER> getAllIfs() Returns a list of if-else statement numbers, when query evaluator calls if(_,_,_)
LIST<INTEGER> getIfsWithControlVariable(INTEGER v) Returns a list of if-else statement numbers with the control variable v.
BOOLEAN isIfControlVar(INTEGER i, INTEGER v) Returns true if the if-else statement i has the control variable v.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getControlVariablesInIf() Returns a pair of integer lists, where the first list corresponds to the if-else statement number, and the second list corresponds to the variable indexes v, such that in the same index of both lists, pattern if(v,_,_) holds.
LIST<INTEGER> getControlVariablesInIf(INTEGER i) Returns a list of control variables in the form of varIdx which corresponds to the if-else statement i
LIST<INTEGER> getIfFromControlVar(INTEGER v) Returns a list of if-else statements that has a control variable v
Calls API
BOOLEAN isCalls(INTEGER p1, INTEGER p2) **BOOLEAN isCalls(STRING p1, STRING p2)** Returns true if Calls(p1,p2) holds
BOOLEAN isCaller(STRING p) Returns true if Calls(p,_) holds.
BOOLEAN isCallee(STRING p)

Returns true if Calls(_,p) holds.
BOOLEAN hasCalls() Returns true if Calls(_,_) holds
LIST<INTEGER> getCallee(STRING p) **LIST<INTEGER> getCallee(INTEGER p)** Returns a list of procedures p1, such that Calls(p, p1) holds.
LIST<INTEGER> getCaller(STRING p) **LIST<INTEGER> getCaller(INTEGER p)** Returns a list of procedures p1, such that Calls(p1, p) holds.
LIST<INTEGER> getAllCallees() Returns a list of procedures p, such that Calls(_, p) holds.
LIST<INTEGER> getAllCallers() Returns a list of procedures p, such that Calls(p, _) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllCalls() Returns a pair of integer lists containing procedure indexes p1 and p2 in the form of procedure indexes, such that in the same index of both lists, Calls(p1, p2) holds.
Calls* API
BOOLEAN isCallsStar(INTEGER p1, INTEGER p2) **BOOLEAN isCallsStar(STRING p1, STRING p2)** Returns true if Calls*(p1,p2) holds
BOOLEAN isCaller(STRING p) Returns true if Calls*(p,_) holds.
BOOLEAN isCallee(STRING p) Returns true if Calls*(_,p) holds.
BOOLEAN hasCalls() Returns true if Calls*(_,_) holds
LIST<INTEGER> getCalleeStar(STRING p)

LIST<INTEGER> getCalleeStar(INTEGER p) Returns a list of procedures p1, such that Calls*(p, p1) holds.
LIST<INTEGER> getCallerStar(STRING p) **LIST<INTEGER> getCallerStar(INTEGER p)** Returns a list of procedures p1, such that Calls*(p1, p) holds.
LIST<INTEGER> getAllCallees() Returns a list of procedures p, such that Calls*(_, p) holds.
LIST<INTEGER> getAllCallers() Returns a list of procedures p, such that Calls*(p, _) holds.
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllCallsStar() Returns a pair of integer lists containing procedure indexes p1 and p2 in the form of procedure indexes, such that in the same index of both lists, Calls*(p1, p2) holds.
Next API
BOOLEAN isNext(INTEGER s1, INTEGER s2) Returns true if Next(s1, s2) holds.
BOOLEAN isExecutedBefore(INTEGER s) Returns true if Next(s, _) holds.
BOOLEAN isExecutedBefore(INTEGER s) Returns true if Next(_, s) holds.
BOOLEAN hasNext() Returns true if Next(_, _) holds.
LIST<INTEGER> getExecutedBefore(INTEGER s, ENTITY type) Returns a list of statement number s1 such that Next(s1, s) holds and s1.type == type.
LIST<INTEGER> getExecutedAfter(INTEGER s, ENTITY type) Returns a list of statement number s1 such that Next(s, s1) holds and s1.type == type.
LIST<INTEGER> getAllExecutedBefore(ENTITY type)

Returns a list of statement number s1 such that Next(s1, _) holds and s1.type == type.
LIST<INTEGER> getAllExecutedAfter(ENTITY type) Returns a list of statement number s1 such that Next(_, s1) holds and s1.type == type.
PAIR<LIST<INTEGER>, LIST<INTEGER>> getAllNext(ENTITY type1, ENTITY type2) Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both lists, Next(s1,s2) holds and s1.type == type1, s2.type == type2.
Next* API
BOOLEAN isNextStar(INTEGER s1, INTEGER s2) Returns true if Next*(s1,s2) holds.
BOOLEAN isExecutedBefore(INTEGER s) Returns true if Next*(s, _) holds.
BOOLEAN isExecutedBefore(INTEGER s) Returns true if Next*(_, s) holds.
BOOLEAN hasNext() Returns true if Next*(_, _) holds.
LIST<INTEGER> getExecutedBeforeStar(INTEGER s, ENTITY type) Returns a list of statement number s1 such that Next*(s1, s) holds and s1.type == type.
LIST<INTEGER> getExecutedAfterStar(INTEGER s, ENTITY type) Returns a list of statement number s1 such that Next*(s, s1) holds and s1.type == type.
LIST<INTEGER> getAllExecutedBeforeStar(ENTITY type) Returns a list of statement number s1 such that Next*(s1, _) holds and s1.type == type.
LIST<INTEGER> getAllExecutedAfterStar(ENTITY type) Returns a list of statement number s1 such that Next*(_, s1) holds and s1.type == type.
PAIR<LIST<INTEGER>, LIST<INTEGER>> getAllNextStar(ENTITY type1, ENTITY type2) Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both lists,

Next*(s1,s2) holds and s1.type == type1, s2.type == type2.

Affects API
BOOLEAN isAffects(INTEGER s1, INTEGER s2) Returns true if Affects(s1,s2) holds
BOOLEAN isAffector(INTEGER s) Returns true if Affects(s,_) holds
BOOLEAN isAffected(INTEGER s) Returns true if Affects(_,s) holds
BOOLEAN hasAffects() Returns true if Affects(_,_) holds
LIST<INTEGER> getAffectedOf(INTEGER s) Returns a list of statement number s1 such that Affects(s,s1) holds
LIST<INTEGER> getAffectorOf(INTEGER s) Returns a list of statement number s1 such that Affects(s1,s) holds
LIST<INTEGER> getAllAffected() Returns a list of statement number s1 such that Affects(_,s1) holds
LIST<INTEGER> getAllAffector() Returns a list of statement number s1 such that Affects(s1,_) holds
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllAffects() Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both lists, Affects(s1,s2) holds
Affects* API
BOOLEAN isAffectsStar(INTEGER s1,INTEGER s2) Returns true if Affects*(s1,s2) holds

BOOLEAN isAffector(INTEGER s) Returns true if Affects*(s,_) holds
BOOLEAN isAffected(INTEGER s) Returns true if Affects*(_,s) holds
BOOLEAN hasAffects() Returns true if Affects*(_,_) holds
LIST<INTEGER> getAffectedStarOf(INTEGER s) Returns a list of statement number s1 such that Affects*(s,s1) holds
LIST<INTEGER> getAffectorStarOf(INTEGER s) Returns a list of statement number s1 such that Affects*(s1,s) holds
LIST<INTEGER> getAllAffected() Returns a list of statement number s1 such that Affects*(_,s1) holds
LIST<INTEGER> getAllAffector() Returns a list of statement number s1 such that Affects*(s1,_) holds
PAIR<LIST<INTEGER>,LIST<INTEGER>> getAllAffectsStar() Returns a pair of integer lists containing statements s1 and s2, such that in the same index of both lists, Affects*(s1,s2) holds

Others

VOID clearCache() Clears all the data stored in the cache.
HASHMAP joinMap (HASHMAP firstMap,HASHMAP secondMap) Combines two maps and their key and values together.

PQL API

void run()

Runs the QueryProcessor