School of Computing

CS3203 Software Engineering Project

AY21/22 Semester 2

Project Report

Team 39

Team Members	Student No.	Email
Lauren Lee Hui Ying	A0204972A	lauren.lee@u.nus.edu
Lim Zheng Wei	A0180324N	limzhengwei@u.nus.edu
Loh Guo Jun	A0200809M	e0407790@u.nus.edu
Tan Xi Zhe	A0200766J	tanxizhe@u.nus.edu
Wang Zhenlin	A0187381W	e0322965@u.nus.edu
Zhu Yuxuan	A0204786X	e0424709@u.nus.edu

Consultation Hours: Tuesday 5 - 6 pm

Tutor: Zhang Xinyi

Table of Contents

Abstract

A Static Program Analyzer (SPA) is an interactive tool that automatically answers queries about programs. Our team designed and implemented a SPA for the SIMPLE programming language which covers all the basic and advanced requirements of SPA. In this report, we will explain our implementation details and justify the different design decisions we have made in this project. This report will also cover our testing strategies and explain our test cases used in Unit, Integration and System testing.

First, we did not implement and make use of an Abstract Syntax Tree for the parsing of the SIMPLE source program. Although AST was one of the first data structures which was introduced during the lectures, our team decided against using AST as the performance was not as efficient compared to other alternatives.

Moving on to pattern matching, for the pattern clause to run efficiently, we parsed all valid substrings as candidate patterns of each assignment and stored them into the PKB. This way, we only need O(1) time to find the assignments with substrings matching the pattern specified in the query. To ensure the substrings are valid patterns for the assignment, we did not implement the substrings as AST themselves, as this would result in nested comparisons with higher time complexity (at least O(log(n)) during the query time (see point 1). The postfix string conversion method solves this problem by using stacks to convert the infix pattern substrings in the assignment statements into a postfix form that simulates the AST perfectly. All valid pattern substrings are generated and stored into the PKB during the postfix conversion process.

Moreover, for populating the relationships into the PKB, we decided to store and populate the reverse relationships as well into the PKB. For example, while the SP only calls the API to store the Parent(a, b) relationship into the PKB, the PKB will also store the reverse relationship where the child is stored as the key with the parent stored as the corresponding value in a separate table. While it was not strictly necessary to do so, we implemented it this way to further reduce the time needed for queries, albeit at the expense of more memory.

Finally, the query parser will do one-pass parsing as it is a faster approach although the implementation will be more complicated. For the query evaluator, we will cache the relationships that are not precomputed to increase the evaluation speed.

Part 1 – Technical Report

1 SPA Design

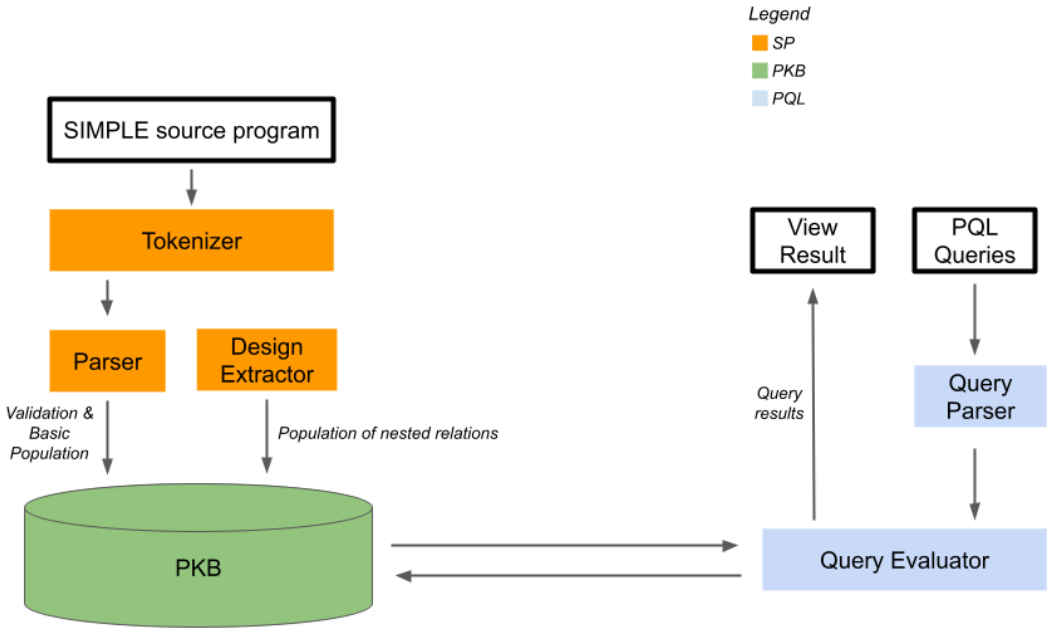

Diagram 1.1 Component diagram illustrating our SPA design

In SPA, there are 3 main components which are the Source Processor (SP), Program Knowledge Base (PKB) and the Query Processing Subsystem (PQL). Each of these components in turn contains their own smaller subcomponents and together, these form the basis of SPA.

For any input SIMPLE source program, the program will first be tokenized by the Tokenizer subcomponent in the SP. Then, the Parser subcomponent will use the list of tokens returned by the Tokenizer to perform Validation and Population of Basic Relationships (ie. UsesP, ModifiesS, Calls, Parent, Follows, Next) and Entities into the Pkb. If the input SIMPLE source program is deemed to be invalid during the validation process, the PKB will not be populated and the parser will throw an InvalidSyntaxException or InvalidSemanticException. The exception will be caught in the TestWrapper, and the program will terminate gracefully. Once the basic relationships and entities have been populated in the PKB, the Design Extractor subcomponent will then interact with the PKB to populate the more complex nested relationships. At this stage, apart from Next*, Affects and Affects* relations, the PKB is fully populated with all the relevant relationships and entities and the SPA program is ready to answer the given input queries.

When a query input comes in, the input will first be handled and parsed by the PQL parser subcomponent. Within this subcomponent there are multiple classes handling the different aspects of parsing (but it suffices to know that the PQL parser subcomponent handles the parsing at the high level). If the query input is deemed to be invalid, the parsing of the query

will be terminated and nothing will be returned to the output and the next query input will then be parsed. If the query input is valid, it will then be handled by the query evaluator subcomponent. This subcomponent contains multiple classes that deal with different aspects of evaluation and in general, the query evaluator subcomponent will interact with the APIs of the PKB to obtain the desired information. Finally, the results of the query input will be returned for viewing.

The above is a high level view of how the different components and subcomponents interact and these will be explained further in their respective subsections 1.2 (Source Processor), 1.3 (PKB), 1.4 (Query Parser) and 1.5 (Query Evaluator).

1.1 Sample SIMPLE program

We will use the following sample SIMPLE program as a running example throughout the report to explain our SPA implementation. Line numbers are not part of the code.

```
   procedure teamReport {
1      countDays = 99;
2      read read;
3      while ((x != 0) && (y > 45)) {
4          if ((monday - tuesday) == (1 % 2)) then {
5              while (x < y) {
6                      x = x + y;
               }
           } else {
7              tuesday = tuesday % count;
8              friday = friday * count;
           }
9          print = x + 1;
10         read y;
       }
11     if (!(thursday <= 0)) then {
12         wednesday = 1 + x + (y * 3);
       } else {
13         thursday = monday / count;
       }
14     print abc123;
15     call meeting;
   }

   procedure meeting {
16     tuesday = friday * 2;
17     friday = tuesday * 4;
18     if (tuesday == 2) then {
```

```
19          if (wednesday == 3) then {
20                call p4;
            } else {
21                print friday;
            }
22          print = monday * 2;
        } else {
23          tuesday = monday * sunday;
        }
    }

    procedure p4 {
24      tuesday = friday + monday;
25      print tuesday;
26      call p10;
    }

    procedure p10 {
27      call p12;
    }

    procedure p12 {
28      monday = monday / 2;
    }
```

1.2 Source processor

For the Source Processor (SP) subcomponent, we will parse and validate an input SIMPLE source program, and populate the Program Knowledge Base (PKB). First, the input source program is split into lexical tokens using the Tokenizer class, and the tokens list returned will be used to validate and populate the entities and relationships in the source program.

The parsing algorithm used for the SP is a top down approach, and the validation and population steps are done in the following order:

1. Create Parser object
    a. Tokenize input source program into lexical tokens
    b. Extract Follows, Parent and Next relationships from source program and populate into PKB.
    c. Generate the Control-Flow Graph (CFG) for each procedure and populate into PKB.
    d. Assign statement number and create Statement object for each statement
2. Create Statement objects
    a. Validate statement
    b. Statements inside the same procedure are grouped together and used to create Procedure objects.
3. Create Procedure objects
    a. Validate Procedure
    b. Extract ModifiesP, UsesP and Calls relationships from the statement list of each Procedure. In this step, only the variables directly used/modified by the procedure will be populated.
4. Validate Procedure Calls
    a. Check if all procedures are calling another procedure which exists in the source program
    b. Check for recursive/cyclical calls
5. Populate Entities in Procedures
6. Populate Entities, Patterns and Uses/Modifies relationships in Statements
7. Populate Nested Relationships
    a. Parent*, Follows*, Calls*, indirect ModifiesP and UsesP

The implementation of each step is explained in more detail in the following sections.

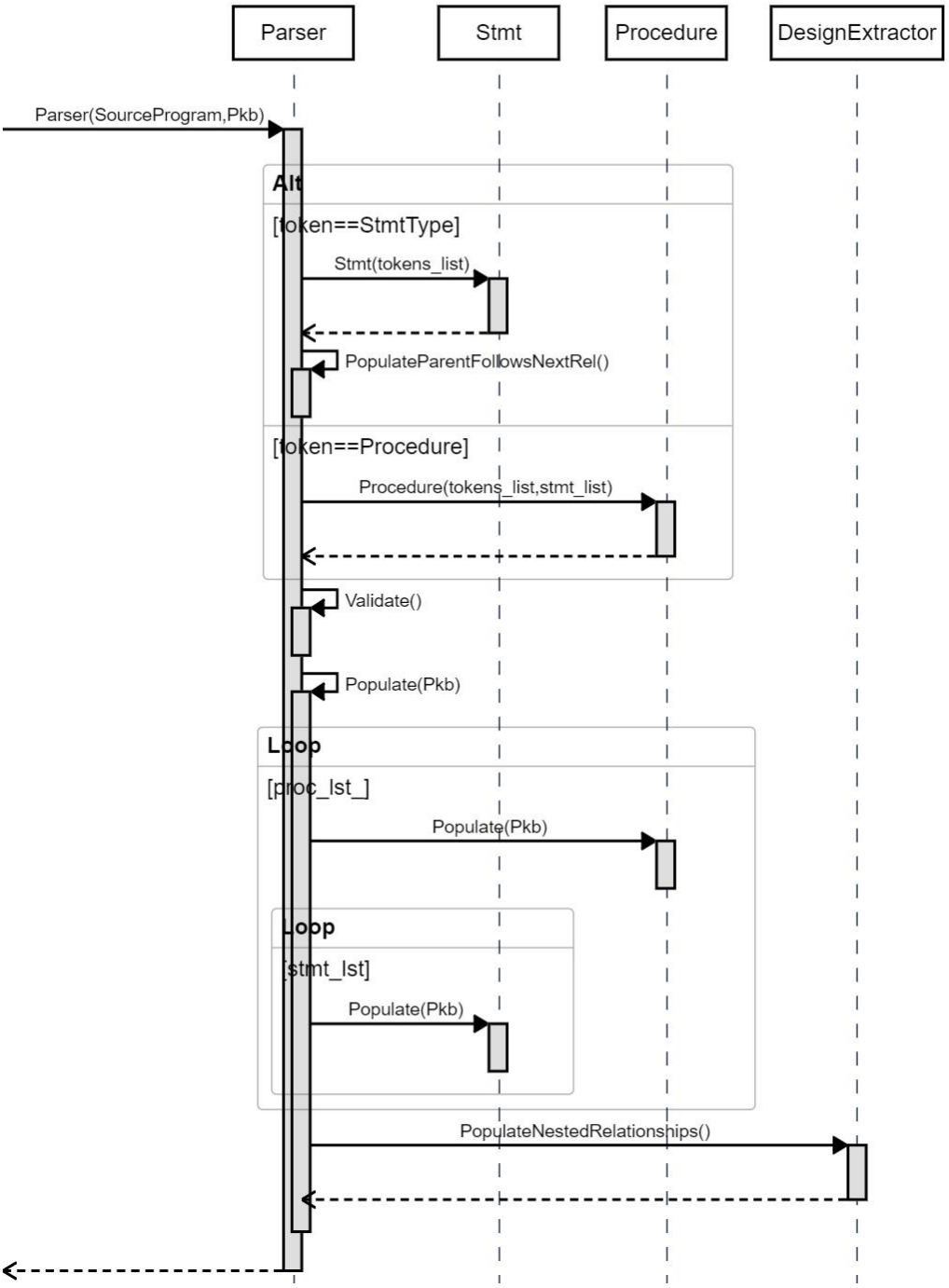

Diagram 1.2.1 Sequence diagram for Source Processor

Tokenizer

The Parser calls Tokenizer which will break up the input SIMPLE program into lexical tokens. The tokenization algorithm is described as follows:

1. Initialise a `vector<Token>` which is a list that will contain completed tokens.
2. Iterate through each character of the source program.
3. For each character, maintain a Token (current token).
    a. Check the type of the current character found (Digit, bracket, whitespace, operator, semicolon, or letter).
    b. If a digit, operator or letter is found, append the current character to the current token. Update the current token with the correct type and text found. (eg. A token with more than one 'Letter' character will be updated to have the type 'Name'.)
    c. If a bracket, whitespace, or semicolon is found, end the previous token by adding the completed token to the `vector<Token>`. A new current token is initialised and is updated with the correct type and text.
4. Repeat step 3 until the end of the source program.
5. Return the list of tokens.

The result of tokenization is a `vector<Token>`, a list of tokens where each token has a class attribute of type and text. For instance, line 14 in the sample SIMPLE program would be broken down into three tokens: (NAME, "print"), (NAME, "abc123"), and (SEMICOLON, ";").

In step 3, if any other character is found besides those specified in the grammar, it will be considered invalid. In step 3b, if a token is created with digits followed by letters, it will also be considered an invalid token. In both cases, it is not possible to create a token, hence an `InvalidSyntaxException` will be thrown.

In line with the Single Responsibility Principle, the Tokenizer does not perform any other validation checks or assign statement numbers, as these will be done in the Parser.

Parser

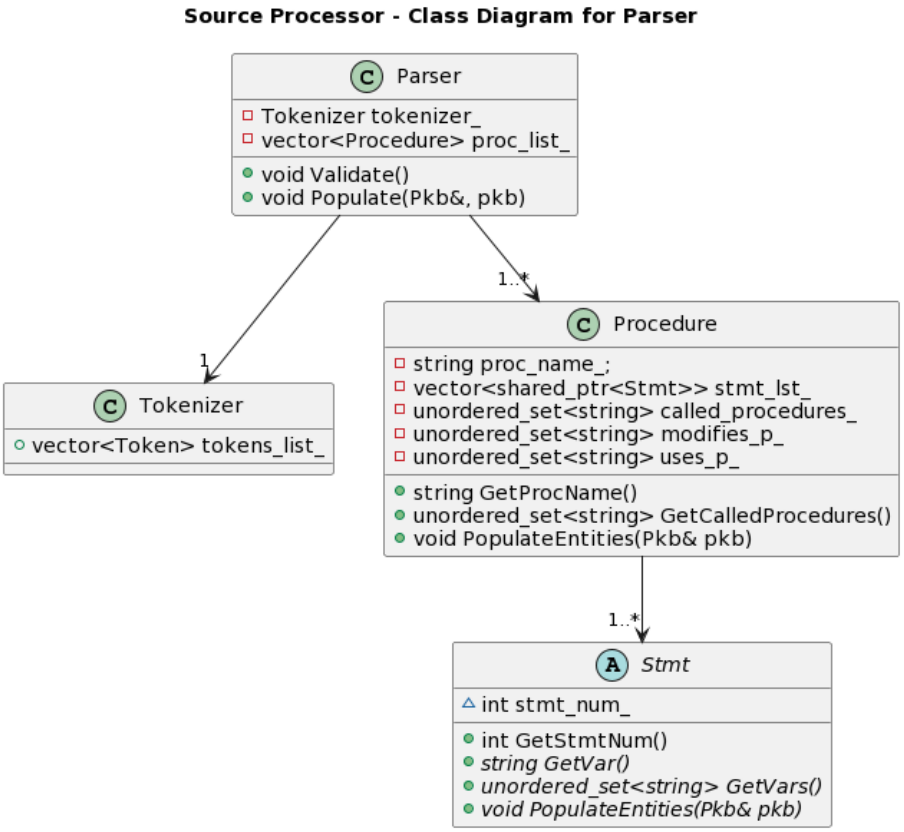

Diagram 1.2.2 Class diagram for Parser

Creating Procedure and Statement Objects

When the parser is iterating through the tokens list returned by the Tokenizer, it will check the token type of the current token and determine the subsequent action taken according to the following logic represented in a flow chart:

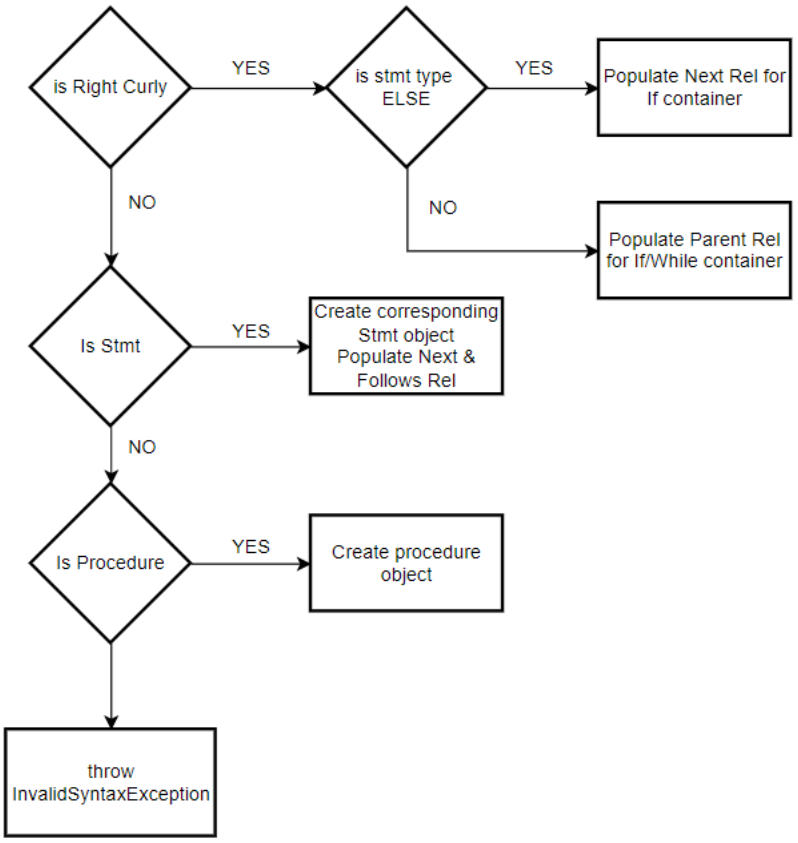

Diagram 1.2.3 Logic of iterating through the tokens list in Parser

A special case to take note of is the assignment statement, which does not have a specific terminal keyword at the start of the statement. To ensure that statements such as `print = x + 1;` are recognised as an assignment statement rather than a print statement, all statements will be checked for their 2nd token (is '=' or not) before determining the statement type. Similarly, the parser will check for statements before procedure declarations to ensure that a statement object will be created for `procedure = 1;` instead of creating a procedure object.

Algorithm for the extraction of Parent Relationship

To extract the Parent Relationship from the SIMPLE Source Program, the parser uses the stack data structure to keep track of the Parent and Children of each container statement. This allows the parser to retrieve the statement numbers of the direct parent and child for each container, which is sufficient information at this stage since the Parent* relationship will be handled by the Design Extractor at a later stage.

Since each container statement may contain multiple statements inside, the data structure used to keep track of the children is `stack<vector<int>>`, while `stack<int>` is used for the parents since we can only have one parent but multiple children. The statement number of each if/while statement will be pushed on top of the Parent stack and the statement number of each statement inside the container will be appended to the vector at the top of the stack.

Using the example program in Section 1.1, by the time the parser reaches line 8, the parent stack will contain {3, 4} and the children stack will contain {{4}, {5, 7, 8}}. After line 8, the parser will recognise the right curly bracket as the end of the if container at line 4 (top of the parent stack) and it will populate the parent relationship for stmt# 4 and its children stmt# 5, 7, 8 (top of the children stack). Then, the top of these 2 stacks will be popped to allow the addition of the subsequent statement numbers inside the while container at line 3.

Algorithm for the extraction of Follows Relationship

To extract the Follows Relationship from the SIMPLE Source Program, another stack data structure is used to keep track of the previous statement numbers (`stack<int>`). The statement number of each if/while statement will be pushed on top of the previous stack and each statement will replace the value at the top of the stack after populating its Follows relationship.

In the same scenario given above, the previous stack will contain {3, 4, 7} when the parser reaches line 8. The parser will then populate the Follows(7, 8) relationship, pop the value 7 and push the value 8 on top of the stack. After line 8, the parser will recognise the right curly bracket as the end of the container and pop the value at the top of the stack, which ensures that line 9 has a previous number of 4 instead of 8.

Algorithm for the extraction of Next Relationship

The extraction of the Next Relationship is similar to the extraction of the Follows Relationship for non container statements (Read/Print/Call/Assign), as both relationships depend on the previous statement number which is kept tracked in the ‘previous’ stack mentioned above. However, there are 2 new cases which the parser need to keep track of:

1. The next statement number of the last statement in a while loop is the statement number of that while loop itself
2. The next statement number of the last statement in an if/else container is the first statement number reached after exiting the else container.

However, the extraction of Next relationship for container statements (If/While) is more complex since both have 2 possible next statement numbers.

The two next stmt numbers for If:

1) Stmt num of If stmt + 1
2) First stmt number in the else container

The two next stmt numbers for While:

1) Stmt num of While stmt + 1
2) First stmt num reached after exiting the while loop

The data structure `pair<int, int>` is used to keep track of all while statements, where the first int is the statement number of the while statement, and the second int is the statement number of the next statement of that while statement (ie. while statement number + 1). For all other statement types, the second int will be represented as 0. By using `pair<int, int>`, the parser will be able to differentiate between while statements and other statements types and populate the next relationship correspondingly, since it needs to wait for the 2nd next statement number of that while to be reached before populating both 'next' statement numbers at once.

Using the SIMPLE source program in Section 1.1 as an example, when the parser reaches line 19 of the procedure 'meeting', it will first push line 19 on a stack of integers named `if_stmt_num`. Then, using another data structure `stack<vector<pair<int, int>>>` named `last_stmt_nums_in_if`, the pair {20, 0} will be pushed on top of the stack. This 2nd stack is used to keep track of all the last statement numbers inside each if container, which can potentially go up to a large number in the event where the statement list inside an if/else container continues branching off using more nested if containers.

At line 21, the parser will recognise that it is inside an else container and the next relationship for line 19 (Next(19, 20) and Next(19, 21)) will be populated. Then, the pair {21, 0} will be pushed on top of `last_stmt_nums_in_if`.

At line 22, the program exits the if container, so the parser will peek at the top of the `last_stmt_nums_in_if` stack and iterate through all the pairs at the top of the stack to populate the Next relationship for all of them (Next(20, 22) and Next(21, 22)).

Generation of CFG

For our SPA, the parser will generate one CFG per procedure in the SIMPLE source program and store it inside the PKB for the PQL to retrieve when handling Affects/Affects* clauses.

To generate the CFG for each procedure, the parser first initializes an empty list named `cfg_tokens`, which is a vector containing `CfgToken` objects. This list will be used to keep track of all the statement types, and the start and end of the procedure and each container statement. The steps taken to create the `cfg_tokens` list are as follows:

1. Initialize an empty stack called `end_tokens`. This stack will contain the strings "if" or "while", which is used to decide the type of token to be added to `cfg_tokens` at each right curly bracket.
2. Add START token to `cfg_tokens`
3. For non container statements (read/print/assign/call), add the corresponding token (READ/PRINT/ASSIGN/CALL) to the list along with its statement number.
4. For container statements (if/while), add the corresponding token (IF/WHILE) to the list along with its statement number. Then, push the statement type onto `end_tokens` ("if" or "while").
5. When a right curly bracket is reached, peek at the top of the `end_tokens` stack and do the following before popping it out of the stack:
    a. If the right curly bracket is for an else statement (ie. } else {), add the THENEND token into the list
    b. If end_tokens.top() = "if", add the ELSEEND token into the list
    c. If end_tokens.top() = "while", add the WHILEEND token into the list
    d. If end_tokens is empty, this marks the end of the procedure and the END token will be added into the list.

After generating the `cfg_tokens` list at the end of each procedure in the source program, the function GenerateCFG will be called and the `cfg_tokens` will be passed to the function. This function will generate the CFG using this tokens list and the CFG returned is populated to the PKB using the `AddCfg` API provided by PKB.

Validation of Procedure Calls

After all the Procedure and Statement objects have been created, the parser will call its `Validate()` function to check if the source program is semantically valid. 2 checks are done during this stage which are as follows:

1. **Each call statement is calling a procedure that exists in the SIMPLE source program**

    The parser has a list of all Procedures in the SIMPLE source program in the attribute `proc_lst_`, and another list of all procedures called by each procedure in the attribute `called_procedures_set_`. For all the procedures in `called_procedures_set_`, the parser will try to find them in `proc_lst_` and will throw `InvalidSemanticException` if it is not found.

2. **There is no recursive or cyclical calls in any procedure**

   To check for recursive and cyclical calls, a directed graph is created where the vertices are the individual procedures and an edge p → q is created if p calls q. Then, we will make use of depth-first search to check for any cycles in the graph. If a cycle is detected, the parser will throw an `InvalidSemanticException` which will stop the parsing.

Procedure

Extraction ModifiesP, UsesP and Calls relationships

The parser will first pass a list of Statement objects which are inside the procedure as an argument to the constructor of that Procedure object. Then, the constructor will iterate through the list of Statement objects to populate the UsesP, ModifiesP and/or Calls relationship according to each statement type.

Statement Type	**Action**
Read	Add variable into `modifies_p_`
Print/If/While	Add variable(s) into `uses_p_`
Call	Add procedure into `called_procedures_`
Assign	Add LHS variable into `modifies_p_` Add RHS variable(s) into `uses_p_`

Validation of Procedures

The validation of the procedure objects is done in the constructor, which takes in a list of tokens and throws `InvalidSyntaxException` if the list of tokens does not follow the grammar rules of SPA. This helps to ensure that all procedure objects created by the parser are valid.

The validation checks put in place are as follows:

- Size of tokens list = 2
- 2nd token in list is a NAME or LETTER

Interaction between SP and PKB for entities and basic relationships

The `populate(pkb)` function in the Procedure class will store the object's procedure name in the referenced PKB, along with its statement number range (start and end) which is used by the Design Extractor to populate the indirect ModifiesP and UsesP relationship.

This function will also call the API provided by the PKB to store the content inside the attributes `uses_p_`, `modifies_p_` and `called_procedures_` into their relevant tables.

Statement

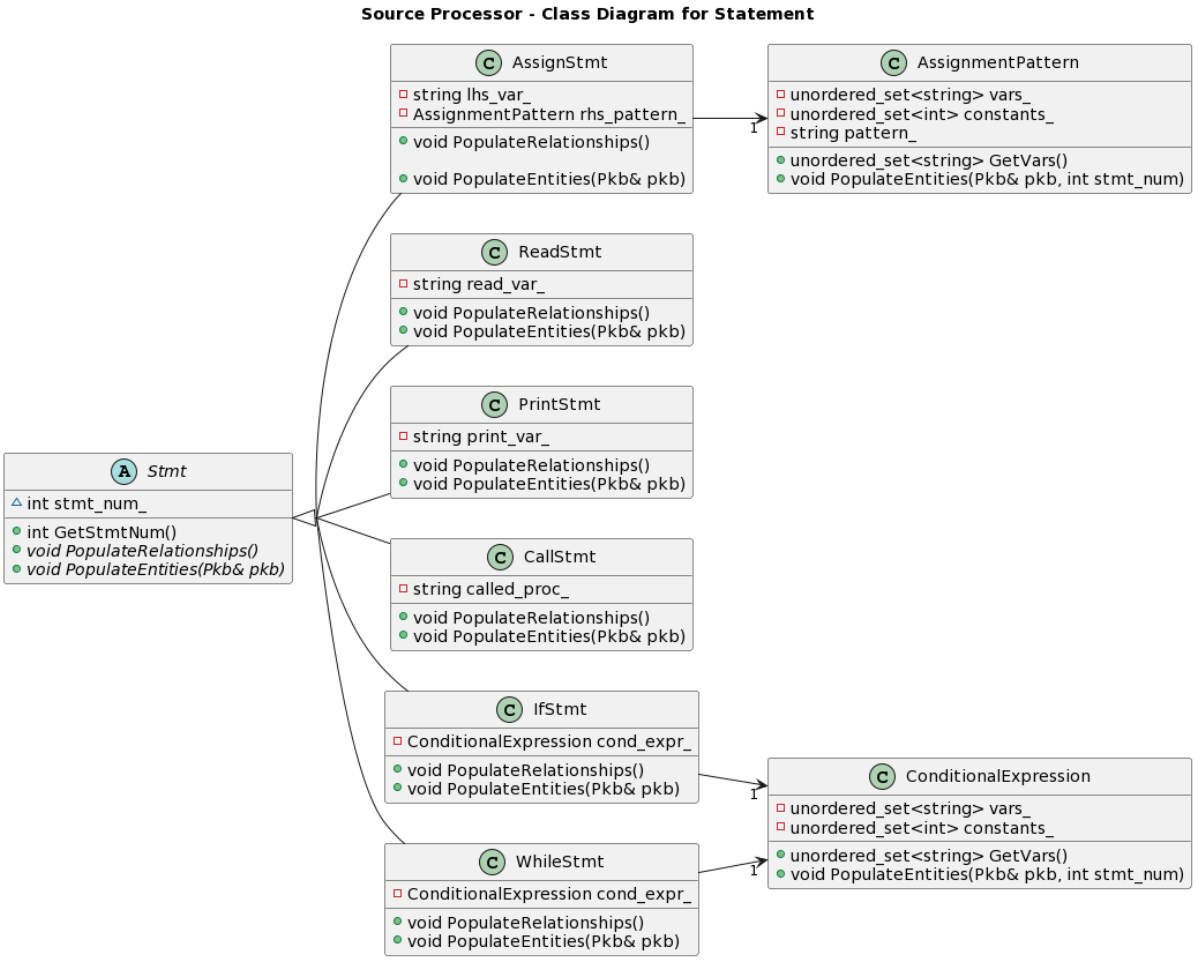

Diagram 1.2.4 Class diagram for Stmt

Design Pattern

The Source Processor used Inheritance to deal with the different behaviors of each statement. An abstract Stmt class with abstract methods acts as the Parent class and each statement type inherits from the Stmt class and implements each method according to its own design abstractions. For example, the function `PopulateRelationships()` for the ReadStmt will be adding its `read_var_` into the ModifiesP relationship while the PrintStmt will be adding its `print_var_` into the UsesP relationship. This design pattern helps us to reduce code duplication as the Procedure can simply call `stmt.PopulateRelationship()` (without knowing the exact type of statement) when iterating through the statement list for that procedure to populate the ModifiesP, UsesP and Calls relationships.

Validation of Statements

The validation of each statement type is done in the constructor, which takes in a list of tokens and throws `InvalidSyntaxException` if the list of tokens does not follow the grammar rules of SPA. This helps to ensure that all statement objects created by the parser are valid.

The table below summarises the validation checks done for each statement type.

Statement Type	**Validation checks**
Read/Print/Call	• Size of tokens list = 2 • 2nd token in list is a NAME or LETTER
Assign	• Size of tokens list >= 3 • 1st token is a NAME or LETTER • 3rd to last token is a valid assignment pattern
If	• Size of tokens list >= 7 • 2nd token is ‘(‘ • 3rd token to 3rd last token is a valid conditional expression • 2nd last token is ‘)’ • Last token is NAME with text ‘then’
While	• Size of tokens list >= 6 • 2nd token is ‘(‘ • 3rd token to 2nd last token is a valid conditional expression • Last token is ‘)’

Validation of Assignment Patterns and Conditional Expressions

For each assignment pattern and conditional expression found in the SIMPLE source program, an AssignmentPattern/ConditionalExpression object will be created by passing a list of tokens to the constructor of each class. Validation is done in the constructor for both types of objects to ensure that the tokens inside the tokens list adhere to the grammar rules of SPA.

To perform the validation of the tokens, the objects will iterate through the list of tokens and create a list of expected tokens for the next token according to the current one. If the next token does not match with any token in the list of expected tokens, the constructor will throw an `InvalidSyntaxException` and the parsing will stop.

The following table shows the mapping of the current token and its expected next token(s) for assignment patterns.

Token Type	**Expected Next Token(s)**
First token	NAME/LETTER/INTEGER/DIGIT/LEFT PAREN
OPERATOR ("*", "/", "+", "-", "%") LEFT PAREN	NAME/LETTER/INTEGER/DIGIT/LEFT PAREN
RIGHT PAREN NAME/LETTER/INTEGER/DIGIT	OPERATOR/RIGHT PAREN

The following table shows the mapping of the current token and its expected next token(s) for conditional expressions.

Token Type	**Expected Next Token(s)**
First token	NAME/LETTER/INTEGER/DIGIT/LEFT PAREN/NOT OPERATOR
REL OPERATOR (">", "<", "==", "!=", ">=", "<=") OPERATOR("*", "/", "+", "-", "%") COND OPERATOR("||", "&&")	NAME/LETTER/INTEGER/DIGIT/LEFT PAREN
NOT OPERATOR ("!")	LEFT PAREN
LEFT PAREN	NAME/LETTER/INTEGER/DIGIT/LEFT PAREN/NOT OPERATOR
RIGHT PAREN	OPERATOR/RIGHT PAREN/COND OPERATOR/ REL OPERATOR
NAME/LETTER/INTEGER/DIGIT	OPERATOR/RIGHT PAREN/REL OPERATOR

Interaction between SP and PKB for entities and Modifies and Uses relationships

The `populate(pkb)` function of each Statement subclass has a different implementation depending on the SPA design abstractions. The function will take in a reference to the PKB table and store all relevant entities and relationships for each statement object using the APIs `AddEntityToSet` and `AddInfoToTable` provided by the PKB team.

The following table shows the interaction between the SP and PKB to add the entities and the Uses and Modifies relationships for each statement type.

Statement Type	**Entities/Relationships Populated**
Read	Add stmt_num_ to Read Entity Set Add read_var_ to Variables Set & Modifies Table
Print	Add stmt_num_ to Print Entity Set Add print_var_ to Variables Set & Uses Table
Assign	Add stmt_num_ to Assign Set Add lhs_var to Variables Set & Modifies Table Add variables(s) in rhs_pattern_ to Variables Set & Uses Table Add constant(s) in rhs_pattern_ to Constant Set
If/While	Add stmt_num_ to If/While Set Add variables(s) in cond_expr_ to Variables Set & Uses Table Add constant(s) in cond_expr_ to Constant Set
Call	Add stmt_num_ to Call Set Add called_proc_ to Caller Table

Interaction between SP and PKB for Assignment Patterns and If/While Patterns

For assignment patterns, the parser on the SP side will add the entire pattern to the AssignPattern Table in the PKB. The PKB will then process the pattern string internally to generate all the possible substrings of the pattern to do matching for PQL queries. The implementation details are explained in Section 1.3 PKB.

For pattern matching for if and while statements, the parser will generate a list containing all the variables used in the conditional expression and add it into the kIfPattern and kWhilePattern tables in the PKB respectively.

Extraction Process by Design Extractor for Nested Relationships

Once the basic relationships (i.e. Parent/Follows/Modifies/Uses/Calls) have been populated into the PKB, the nested relationships can then be populated using these basic relationships. To do so, the Design Extractor will retrieve the relevant tables from the PKB using the public APIs provided.

In one of the above sections, the Parent population was explained and used as an example and once the process is done, the PKB will contain the following Parent relations for the procedure teamReport.

Parent (Key)	Children (Values)
5	6
4	5, 7, 8
3	4, 9, 10

Using the Parent relationships retrieved from the PKB, the Design Extractor will then populate the transitive Parent* relationships into the PKB and the following Parent* relations will be stored into the PKB. This process is then repeated for the other nested relationships such as Calls*, Follows*, nested Uses and nested Modifies.

Parent* (Key)	Children (Values)
5	6
4	5, 7, 8, 6
3	4, 9, 10, 5, 7, 8, 6

During the nested population process, there are certain nested relationships that must be populated first as other nested relationships depend on it. For instance, the nested relations for Uses and Modifies depends on the nested relationships for Parent* and it needs to be completed first. If the Parent* relations is not populated first, it will result in an incomplete set of relationships for nested Uses and Modifies, thereby leading to incorrect results for the PQL input queries.

1.2.1 Design decisions

Design consideration #1: AST is not implemented in our SPA

Our team decided not to use an AST to derive the Follows, Parent relationships and pattern matching. We used tables to store these 3 information instead, as it is easier to implement and also performs better compared to an AST during retrieval. A more detailed analysis of the comparisons made is listed in the table below.

Criteria	Implement AST for Follows, Parent, pattern	Store Follows, Parent, pattern in PKB tables
Complexity of implementation	More complex. Each Tree Node in the AST can have 0 to an infinite number of children nodes, which makes the structure of the tree unpredictable and hard to debug. Traversal of the tree will also be more complex than directly getting the values from the PKB tables.	Less complex. Easier to debug and check if the relationships and patterns are correctly populated inside the PKB tables with the help of existing functions inside the C++ Standard Library.
Performance at retrieval	Worst case: O(n) time. Since the AST can be an unbalanced tree, tree traversal all the way down to the leaf nodes can take up to O(n) time.	O(1) time. For pattern matching, we are storing all the possible substrings in the table so no further computation is needed during retrieval.
Extendibility to Iterations 2 and 3	Unable to extend to future iterations. The 2 new relationships (Next and Affects) introduced in the Advanced SPA cannot be captured in an AST so it can only be used for Follows, Parent and pattern.	The Follows, Parent and pattern tables themselves cannot be extended to include the Next and Affects relationships, but the generic tables which they had inherited from can be extended to them.
Workload	Parallel development for Source Processor and PQL will be harder. Since there is no existing data structure for the implementation of the AST, the Source Processor team needs to implement and build the AST first before PQL querying and retrieval can take place.	Parallel development is easier as the data structures used to store the information in the PKB tables are similar to the ones provided in the C++ Standard Library.

Design consideration #2: Tokenize ‘read’, ‘print’, ‘if’, ‘while’ as NAME

Initially, we considered giving each statement their own token type (i.e. ‘READ’, ‘PRINT’ etc). This would be helpful during validation and population, since each statement will be uniquely identified just from their token type and there would be no confusion amongst a ‘read’ statement and ‘read’ variable. However, we realised that additional checks would have to be done during Tokenization, and these checks overlap with the checks done during Validation. Hence, we decided to adhere to the Single Responsibility Principle, which is to let the Tokenizer parse and return all the inputs as generic names first before checking the data inside each token during validation.

Design consideration #3: Populating Parent, Follows, Next during the constructor of Parser object instead of Procedure object.

We considered populating the Parent, Follows and Next relationships in the constructor of the Procedure object inside of the constructor of the Parser object since these relationships are independent among all the procedures in the SIMPLE source program. However, this would mean that the ‘right curly bracket’ tokens will need to be passed to the Procedure’s constructor, since they are used to keep track of when each container statement ends. This means that the Procedure object will have to iterate through the whole tokens list again, hence we decided to extract these relationships out earlier so that the whole tokens list can be discarded after one iteration and the Procedure constructor only needs to take in the statement list to perform further abstractions.

Design consideration #4: Using DFS instead of BFS in Design Extractor for populating certain nested relationships into PKB

For iteration 1, we mainly focused on correctness and as such, the BFS algorithm used for populating nested relations such as Parent* and Calls* was not optimised in terms of efficiency. From the sample running source program defined, using the initial algorithm used for iteration 1, assuming that the Calls* relation is populated for procedure teamReport first, the algorithm would then visit procedures meeting, p4, p10 and then p12. Next, if the algorithm were to populate Calls* for procedure meeting, it would visit p4, p10 and P12 again and this is not efficient as the same “nodes” were visited multiple times.

To improve the efficiency of the population algorithm, from iteration 2 onwards, we updated our algorithm to that of DFS. Using this algorithm, if the algorithm were to populate procedure ‘teamReport’ first, it would visit procedures ‘meeting’, ‘p4’, ‘p10’ and ‘p12’. If it were to populate ‘meeting’ next, it would realise that ‘meeting’ has already been visited and the Calls* relation for ‘meeting’ has already been populated and there is no longer a need to recurse further on that procedure. As such, this new algorithm eliminates the need to visit the nodes multiple times.

1.2.2 Abstract API

The most important APIs provided by Source Processor are listed below.

API	**1-2 line description**
BOOL Validate()	Implemented by Parser. Checks if the procedures in the SIMPLE input source program are semantically correct.
VOID populate(PKB pkb)	Implemented by Parser. Interacts with the PKB and populates all entities and basic relationships such as Parent, Follows and Calls.
VOID PopulateNestedRelationships()	Implemented by DesignExtractor. Interacts with the PKB to populate all nested relationships.
void PopulateForF(TABLE to_refer, TABLE to_update)	Implemented by DesignExtractor. Interacts with the Follows Table in the PKB to populate the Follows* Table.
void PopulateForPOrC(TABLE to_refer, TABLE to_update)	Implemented by DesignExtractor. Interacts with the Parent/Calls Table in the PKB to populate the Parent*/Calls* Table.
void PopulateNestedModifiesOrUses(TABLE to_refer, TABLE to_update)	Implemented by DesignExtractor. Interacts with the Parent* Table in the PKB to populate the nested Uses or Modifies relationship for each if/while statement.
void PopulateNestedModifiesPOrUsesP(TABLE to_refer, TABLE to_update)	Implemented by DesignExtractor. Interacts with the Calls* Table in the PKB to populate the nested Uses or Modifies relationship for each procedure.

The full list of APIs provided by SP are in the appendix (Section 8.1 SP Abstract API).

1.3 PKB

A custom table class wrapper using hash maps as the underlying implementation was used as the main data structure for storing relationships in the PKB. This table was implemented as a **template** such that it could be instantiated with different types for the keys and values while reducing the need for code duplication. For example, in storing the `Follows(a, b)` relation, a relationship table RelTable with both key-values having an int type was created using the table template and this allowed us to store the mapping from an integer to another integer. This RelTable is a parent class to all individual relationship tables including FollowsTable. Defining RelTable allows us to implement functions applicable to all tables storing relationships which reduces code duplication significantly.

On the other hand, defining the Follows relationship as a class FollowsTable allows us to extend operations specific to the Follows tables. In our implementation, we created a table for the `<stmt_no_1, stmt_no_2>` pair which stores `stmt_no_1` as key and `stmt_no_2` as value if `Follows(stmt_no_1, stmt_no_2)` holds. A table would be declared using

```
class RelTable: TableTemplate<INTEGER, INTEGER>

class FollowsTable : public RelTable
```

and it would store the following as an example:

INTEGER stmt_no_1	INTEGER stmt_no_2
1	2
2	3
4	5

For these tables, the PQL is able to obtain which statement follows a specific statement number to verify if `Follows(a, b)` holds with specified a and b, as well as to retrieve key-value pairs directly satisfying certain conditions for instance.

In order to store entities, a hash set was used as opposed to a table. As an example, a set was used to store all the procedure names. Using a hash set is also convenient as we can remove duplicates easily via the hash function. In C++, an unordered set gives features that allow insertion of duplicate entries to be ignored. This facilitates our implementation as well. For example, a set storing variable names can contain the following:

STRING var_name
“x”
“procedure”

Using the above set, we can do insertion and check if a string is a variable name by lookups with O(1) time.

Table types in PKB

For various design abstractions, we used three classes RelTable, RelListTable and RelListReverseTables which uses inheritance from the template Table discussed above. These three classes are storing integer mappings where the integers either represent the statement numbers or the indexes of variable/procedure names. RelTable stores a one-to-one mapping that represents design abstractions with bijective properties like Follows. RelListTable stores a one-to-many mapping that represents design abstractions like Parent where the parent statement can have multiple child statements or Uses where a statement may use multiple variables. RelListReverseTable is helpful when the query evaluator asks for reverse relationships such as getting all statements `s` in `Uses(s, v)` with a known variable `v`. This table is also a one-to-many mapping. However, it overrides the original table's UpdateKeyValuePair function as the reverse relationship is populated using the same set of parameters passed in. For example, using the statement number as keys and the list of variable indices it uses as values, we can store the direct Uses relationship in RelListTable and the reverse Uses relationship in RelListReverseTable using different implementations of UpdateKeyValuePair functions.

Patterns Storage

For patterns, we are using a table with pattern string as keys and an unordered set of integers as values to store the pattern string-to-statement number mapping. A table with integers as keys and an unordered set of strings as values is also used to store the statement number-to-valid pattern string mapping. Note that for While and If patterns, the pattern strings are essentially variable names. Hence the population of tables related to If/While are done using the variable names obtained when the source processor parses through the If/While statements (i.e. when it obtains the variables that are used by the If/While statements).

For the assignment patterns, however, we are storing valid pattern substrings via Infix to Postfix conversions. This enables the query evaluator to quickly obtain the results by searching the given pattern among the valid pattern substrings or comparing against the exact valid pattern string. We elaborate more details of this postfix conversion in the next few paragraphs.

The conversion is done with a PatternHelper class using stacks. It can produce the exact postfix pattern or the list of valid postfix pattern substrings through the is_full parameter in the GetPatternSetPostfix API. To separate variable names so that situations like "ab + bc + c" and "abb + c + c" both lead to a postfix "abbc+c+", we chose to add a separator symbol "|"

between variable names when generating the postfix form. The result is then populated into the PKB tables mentioned above, with the integer representing the assignment statements' numbers. The query evaluator can then derive the relevant results by first converting the infix pattern string in the pattern clause into postfix form using PatternHelper, and searching within the table. For example, when we have an assignment pattern "(a+b)+c", the source processor will first generate the list of valid pattern substrings for this pattern as {"a", "b", "c", "a|b+", "a|b+|c+"}. The information is then stored in the three tables as follows:

INTEGER stmt_no	LIST_OF_PATTERN valid_pattern_substrings
1	"a", "b", "c", "a|b+", "a|b+|c+"

PATTERN valid_pattern_substring	LIST_OF_INTEGER stmt_nos_with_the_pattern
"a"	1
"b"	1
"c"	1
"a|b+"	1
"a|b+|c+"	1

PATTERN exact_pattern	LIST_OF_INTEGER stmt_nos_with_the_pattern
"a|b+|c+"	1

Note that in the last table, the key is the exact full pattern for the statements whose assignment pattern's postfix form is exactly the string in the key.

Handling With clause

To handle the with clause, the PKB needs to store information with regard to the attributes of the entities. While the statement's number and variable/procedure names are easily accessible, others like the variable name a print statement uses need additional tables to be kept track of. Luckily we have derived these attributes when populating for Uses/Modifies/Calls relationships in design extractor. Thus the variable/procedure names

can be easily transferred into population attribute tables mapping statement numbers with these values. Note that while we can still use the Uses tables built for the design abstraction part, we choose to design tables specifically storing information for each type of statement to make the implementation for both PKB and query evaluator more coherent and extensible.

These entity tables are built using inheritance from the EntityTables which inherit from various classes of Table template. The Assign/Read/Print/Call attributes are stored via individual tables inheriting from a EntityVarsTable which maps from an integer to a string. This represents the statement number-to-attribute variables/procedure names mapping. For tables storing attributes for if/while statements, it inherits from EntityVarsListTable where the integer-to-list of string mapping represents the statement number-to-used variables mapping. Lastly we justify the last layer of inheritance from EntityVarsTable/ EntityVarsListTable instead of directly using them for each table by mentioning the future extensibility of individual classes. For example, a CallerTable inheriting from the EntityVarsTable may have additional functions like combining two call statements whose procedures are identical whereas other tables like AssignTable cannot do this.

Finally, we do note that we are using pointers to these tables in our PKB class and its function implementations to reduce redundant memory usage caused by table copying.

Interaction with SP

One of the main interactions that the PKB has with the SP is the population of relationships and entities. The SP calls the APIs provided by the PKB to populate the relevant information into the PKB. However, there are certain relationships and information that cannot be populated by the SP on the fly. These relationships include nested relationships such as Follows*, Parent* and Calls* as well as nested Uses and Modifies relationships and these nested relationships would have to be populated after the basic relations have been added. To generate these nested relationships, the PKB would have to provide certain tables to the design extractor in the SP such that they are able to make use of the information to populate nested relationships into the PKB.

To reduce the excessive number of APIs provided to the SP for population, function overloading was also implemented to reduce the complexity of calling different functions from PKB to populate the relationships and entities.

Class Diagram for PKB

Diagram 1.3.1 Partial class diagram for PKB

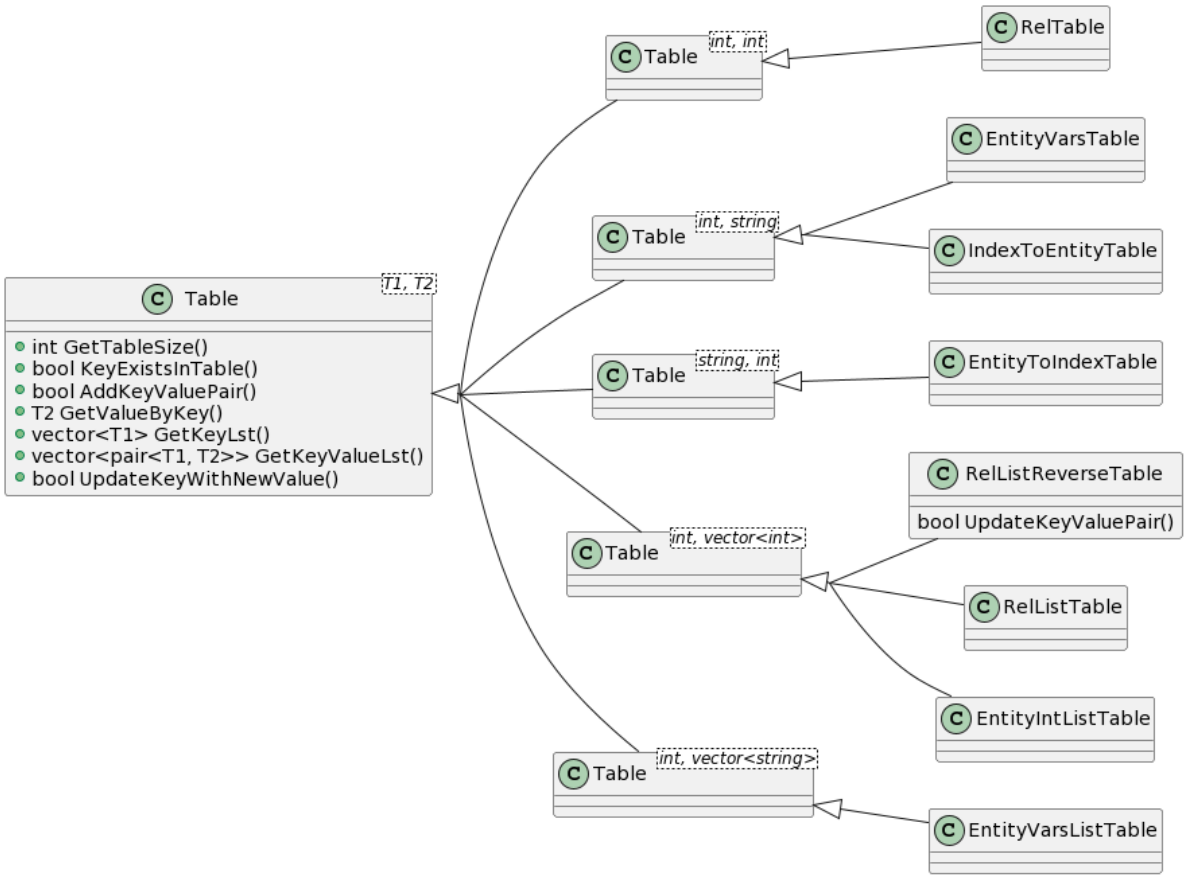

Diagram 1.3.2 Partial class diagram for PKB

1.3.1 Design decisions

Design consideration #1: Using custom tables

After much consideration, our team decided to use a custom table using hash maps as the underlying design for storing relationships in the PKB. While there were other structures such as explicit tree structures (e.g. AST) that were considered during the brainstorming process, it was ultimately deemed unsuitable based on the criteria that our team had set out as mentioned in Section 1.2.1.

Storing relationships using such custom tables allowed the PQL to query and retrieve the desired information in O(1) time whereas storing the AST directly in the PKB meant that the time needed for retrieval of information each time the PKB was queried would be more expensive as a traversal over the tree nodes for the different relationships is required. On the other hand, storing the AST instead of multiple tables would have resulted in a lower memory usage but our team decided to prioritize time over memory.

Using the Parent relationship as an example, if the PKB stored the AST directly, whenever the PQL queries for the children, the AST would need to be traversed to find all its children and this takes O(n) time in the worst case. However, if the Parent relationship was to be stored using a table, retrieval of its children would take O(1) time. Moreover, using tables to store relationships is also easier implementation-wise and less prone to errors.

Design consideration #2: Reverse relationships storage in PKB

Another design consideration was the population of reverse relationships in the PKB. When adding a `Follows(a, b)` relationship into the PKB, the PQL would be able to query whether b follows a in O(1) time. However, getting the reverse relationship would no longer be O(1) as the values in the table would need to be searched and this would require at least O(n) time. In order to reduce the time needed for these queries from the PQL, we decided to also populate the reverse relationships into the PKB, thereby trading memory for efficiency and speed.

Criteria	Populate reverse relationships	Search the values for the inverse relationship
Complexity of implementation	Easier as the relevant value can be retrieved by key	More tedious as the values in the table would need to be searched
Performance at retrieval	O(1) time	O(n) time. The list of values in the table would need to be searched

Design consideration #3: Using only int types for keys and values in the tables

In our implementation for table keys and values, we chose to use indexes to represent entities with non-integer types as this allows the result passed to the query evaluator to be more consistent. More specifically, this design will simplify the implementation in PQL methods as they do not need to worry about type conversions. However, it is still possible for PQL to get the original value of the entities (e.g. variables' name in the form of string) when needed by referring to the index tables in PKB.

With this design for example, we can then send some entries of the Modifies relationship to the PQL where the statement number maps to a variable index (which represents a specific variable string. Taking an example, var_idx 2 might map to the variable "c")

Stmt_no	var_idx
2	2
6	3

and PQL can merge this result with other integer tables without the need to construct special tables that can store variable names as strings. It also allows the PQL component to easily manipulate these data as int type values in C++ are generally easier to work with.

Finally, when the PQL component needs to obtain the variable names mapping corresponding to the index, they are able to do so using the APIs provided by PKB.

var_idx	var_name
2	read
3	x

Design consideration #4: Exposing only relevant tables for Design Extractor

As mentioned in section 1.2, during the population of nested relations by the design extractor, it would require certain tables from the PKB. These tables could have been given public visibility for easier access by the Design Extractor but we decided against it as this would violate the abstraction principle and may lead to accidental or unintended access. As such, these tables were made private and getter methods for the relevant tables were then provided. For tables that were not required by any component, no getter methods were provided. This thus helps to ensure that only the relevant tables are exposed and the others are kept as a black box and it also helps to guard against unintended access.

Design Pattern #1: Inheritance

As seen in the partial class diagram 1.3.2 above, the PKB component stores multiple relationship and entity tables of different types which inherits from a table with a defined type (middle layer) which in turn inherits from a generic table class template. Although the defined type tables (middle layer) are of different types, they share multiple common functionalities and methods such as adding a new key-value pair and using inheritance allows us to define the functionalities once and this follows the DRY principle.

In addition, as seen in the diagram 1.3.2, we have decided to utilise 3 layers of inheritance where the relationship and entity tables would inherit from the tables with defined types (middle layer). We could have made use of only 2 levels of inheritance where the relationship and entity tables would inherit directly from the generic class template. But since many of the tables have the same type, if we needed to extend and support new functionalities for all tables with type `<int, vector<int>>`, we would have to manually add the same functionality in all the other relationship and entity tables of this type. With the middle layer tables and inheritance, it would allow us to support the new functionality by making changes to that single class (open-close principle).

Finally, we also decided to further separate the tables used for storing entities as well as relationships into different types of tables as this would help with extensibility as well as separation of concerns, even if they had the same typing for example RelListTable (short for relation list table) as well as EntityIntListTable. Using this design pattern, if we needed to support new functionalities for say the relationships, we could simply extend the changes in the desired relation table without affecting the entity table.

In short, using inheritance for the tables allowed us to achieve a high degree of separation of concerns in addition to the system following the Open-Close Principle and Single Responsibility Principle.

Design Pattern #2: Condensing duplicate methods into one general method using maps

In iterations 1 and 2, the numerous APIs that were provided to the PQL component for querying about the relationships generally had the same functionalities apart from the fact that these methods were querying from different tables. There were more than 20 APIs provided and in iteration 3, these were condensed to 5 methods.

Using two methods as examples, `IsParent(stmt_1, stmt_2)` and `IsFollows(stmt_1, stmt_2)` had the exact same implementation except that these methods were using different underlying tables. As the number of different relationships grew, the number of methods that had this exact duplication increased and this made it

tedious to maintain the code. As such, in order to reduce the number of methods, mappings were used.

The relationship types that the PQL could potentially query about were stored as the keys in the map, with its value being that of the corresponding table in the PKB. With this, it significantly reduced the code duplication as well as helping with the ease of maintainability and extensibility. If there were new relationships to be added, there would no longer be a need to copy and duplicate the method. Instead, a single line addition to the mapping would suffice to achieve the same functionality.

In addition, this reduced the number of APIs that the PQL component would need to call. The PQL could simply call one general function and pass in the desired relationship type as a parameter to query. The method could also easily be extended to include new relationships without requiring a change in the PQL component.

Refactoring and Optimisation #1: Throwing exceptions instead of returning values

In iterations 1 and 2, there were a few APIs provided that returned either a boolean value or an integer when an exception occurred. For instance, whenever there was an error occurring from the population of tables by the SP, a boolean would be returned as opposed to an exception. However, this was not the best option semantically and there were a few exceptions thrown that were not caught because a boolean value was returned as opposed to an exception.

As such, we decided to update the relevant methods to throw an exception instead of returning special values such as integers or a boolean.

However, certain APIs provided by the PKB are still returning boolean values instead of throwing exceptions as it did not make sense for the method to throw an exception. One such method would be the relationship APIs provided to the PKB. In certain cases where the relationship does not exist, a false boolean value would be returned to the PQL as opposed to an exception.

Refactoring and Optimisation #2: Using memoization

For most of the relationships stored in the PKB, the PKB provides a method to the PQL component to get all possible pairings for that particular relationship to help with the answering of queries. This takes O(n^2) each time to compute all possible pairings. Since this was a method that was called frequently by the PQL component, we decided to memoize the pairings such that it would only be computed for the first time. This uses more memory to store all the possible different pairings for the different relationships but it ensures that subsequent calls of this method would only be O(1) as the result has already been computed.

1.3.2 Abstract API

The most important APIs provided by PKB are listed below.

APIs for SP	**1-2 line description**
BOOL AddInfoToTable(TABLE_IDENTIFIER table_identifier, INT key, LIST_INT value)	Takes in a key and value and adds the information to the relevant table based on the identifier. The internal checking is done by private functions in PKB. Will throw an invalidIdentifierException if the input types do not match the table's key-value types.
BOOL AddInfoToTable(TABLE_IDENTIFIER table_identifier, INT key, LIST_STRING value)	Takes in a key and value and adds the information to the relevant table based on the identifier. Will throw an invalidIdentifierException if the input types do not match the table's key-value types.
BOOL AddEntityToSet(TABLE_IDENTIFIER table_identifier, INT entity_value)	Takes in the entity content and stores it into the respective entity set based on the identifier. Will throw an invalidIdentifierException if the input types do not match the set's entity types.
BOOL AddCfg(CFG cfg)	Takes in a CFG and adds it to a list stored in the PKB

APIs for PQL	**1-2 line description**
BOOL IsRelationshipHolds(RELATIONSHIP_TYPES rel_type, INT stmt_1, INT stmt_2)	Depending on the rel_type passed in, it can check and return whether stmt_1 is a parent of stmt_2 or if stmt_1 calls stmt_2 and so on.
LIST_INT GetRelFirstArgument(RELATIONSHIP_TYPES rel_type, INT second_arg_idx)	Depending on the rel_type passed in, it either returns the parent of second_arg_idx if any or the procedure that called second_arg_idx and so on.
LIST_PAIR_INT GetRelArgumentPairs(RELATIONSHIP_TYPES rel_type)	Depending on the rel_type passed in, return a list of pairs of all the keys and values for a particular relationship type
STRING GetStringByIndex(INDEX_TABLE_TYPE idx_table_type, INT idx);	Takes in the integer index mapping and returns the actual string variable that the number maps to for that particular index table type
INT GetIndexByString(INDEX_TABLE_TYPE idx_table_type, STRING entity);	Takes in the string variable and returns the number index that the variable maps to in the specified index table type

The full list of APIs provided by PKB are in the appendix (Section 8.2 PKB Abstract API).

1.4 Query Parser

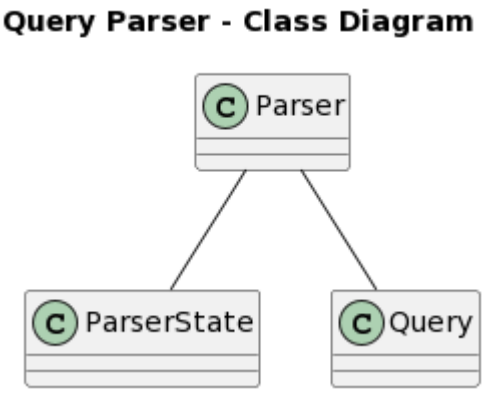

Diagram 1.4.1 Class Diagram for Query Parser

The main subcomponents in Query Parser are represented by the following three classes:

1. ParserState

The ParserState is a relatively simple component that is responsible for reading the PQL query input string when requested by the parser. The parser will decide if the ParserState should expect a certain type of tokens like NAME, INTEGER, brackets etc.

2. Parser

The parser is a component that has a ParserState as the driving component that consumes the query text and generates meaningful tokens. When parsing the query text, the parser will let the ParserState consume a keyword, then expects tokens with respect to the keyword parsed.

For example, when the ParserState consumes a declaration keyword, the parser will expect the ParserState to have one or more synonyms tokenized. When the ParserState consumes a 'Select' keyword, the parser will expect the ParserState to have a synonym tokenized. When the ParserState consumes a 'such that' keyword, the ParserState is expected to tokenize a relationship.Also, when the 'pattern' keyword is consumed, the ParserState is expected to tokenize a pattern. Last but not least, when the 'with' keyword is consumed, the ParserState is expected to tokenize a with clause.

The alternative is collecting all tokens into a collection of tokens and feeding it to the parser. However, this is not chosen because this would let the parser have theoretically unlimited tokens of lookahead, and was deemed unnecessary as the grammar is unambiguous enough to not require such lookaheads.

3. Query Object

A query object is a user-defined class which is built by the parser when parsing the PQL query text. The class diagram below illustrates the hierarchy of Query for the PQL:

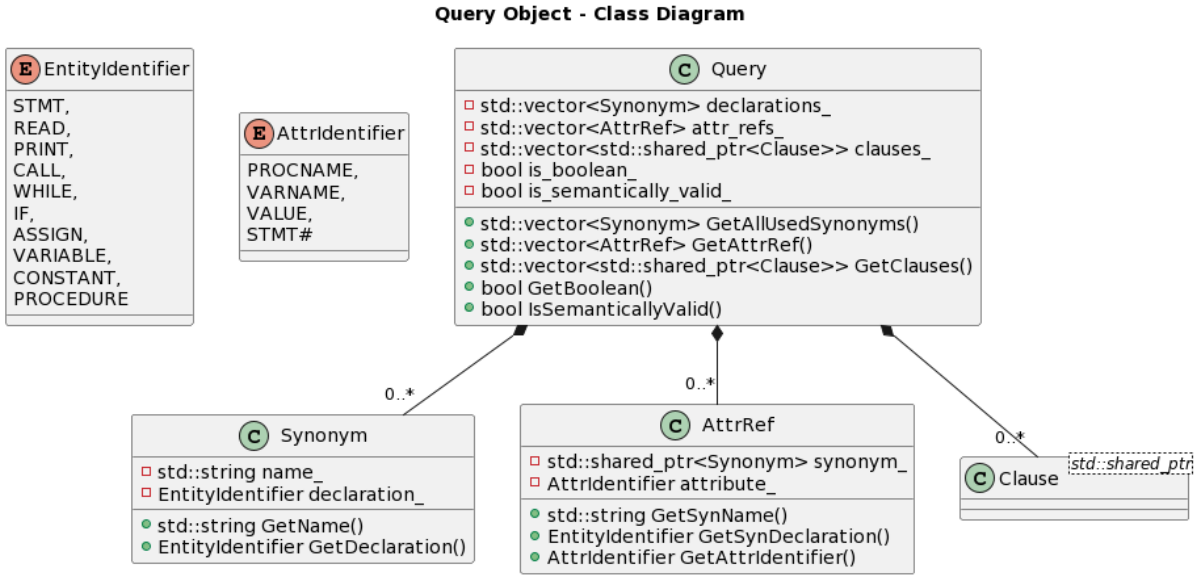

Diagram 1.4.2 Class diagram for PQL Query (Class diagram for Clause will be shown in Query Evaluator)

After the parsing of the query text is done, the query object will be created and passed to the Query Evaluator given the query is syntactically valid. A few APIs will be provided by the Query class for Query Evaluator to extract meaningful information from the query.

Interaction between subcomponents in Query Parser

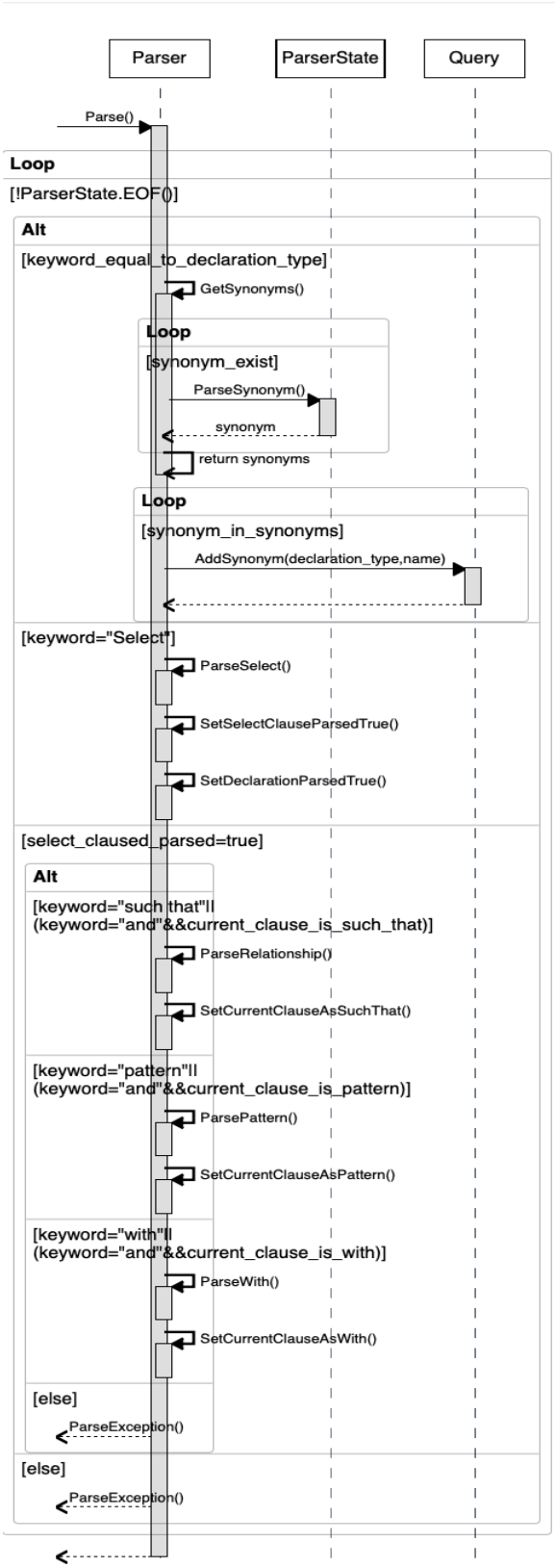

Diagram 1.4.3 Sequence diagram that illustrates the interactions between different subcomponents in PQL query parsing.

PQL Query Validation (Syntactic and Semantic)

Detecting Syntactic Errors

We detect syntactic errors by checking the tokens against the PQL grammar in the parser. Any query that violates the PQL grammar will be handled as a syntactic error.

Handling of the syntactic invalidity:

When the parser detects a syntactic error, the parsing will be terminated and a 'ParseException' will be thrown to indicate that the query is invalid. Since the query will fail eventually, it is unnecessary to continue the parsing to completion. When evaluating the query, we will try the block of parsing and evaluation of the query and catch the 'ParseException' if it is thrown, the query evaluation will not be started if the query is invalid.

Detecting Semantic Errors

We define the following violations as semantic errors (non-exhaustive):

- Use of undeclared synonyms
- Repeated synonym name for a synonym declaration
- Synonym used for pattern clause is not an ASSIGN, IF or WHILE
- The pattern clause syntax does not match the declaration type of the synonym
- First argument for pattern clause is not a VARIABLE synonym if it is not a WILDCARD
- First argument of Modifies and Uses relationship is a WILDCARD
- Second argument of Modifies and Uses relationship is not a VARIABLE
- First argument of Modifies is PRINT
- First argument of Uses is READ
- The type of left and right hand side of a with clause does not match

<u>Use of undeclared synonyms</u>

When the parser is parsing a synonym name in the query, it will check if the name is already declared (this can be done by calling the factory function SynonymDeclared(name) provided by the Query class). If it is not, the query will evaluate to semantically invalid.

<u>Repeated synonym name for a synonym declaration</u>

When the parser encounters a declaration, it will create a new Synonym object and add the new synonym to the list of declarations in the query object. Before adding the new synonym, it will check if the name of the new synonym already exists in the declaration list. If it does, the query will evaluate to semantically invalid.

Checking of the synonym for a pattern clause

Firstly, the parser will parse the pattern clause syntax. Then for each type of pattern clause syntax, there will be a domain of valid synonym type.

Pattern Syntax	**Domain**
syn(_, _)	WHILE, ASSIGN
syn(_, _, _)	IF
syn(_, “expr”)/syn(_, _”expr”_)	ASSIGN

The parser will then check if the type of the synonym is in the valid domain type of the pattern syntax. The query will evaluate to semantically invalid if it is not.

Checking for the argument for Such That clauses

For each type of relationship, there will be a mapping to a set of entities domain that is semantically valid for each argument. The mapping is in the form of `map<RelationshipTypes, unordered_set<EntityIdentifier>>` for the left and right argument.

Hence, the parser will extract the relationship, the left and right argument for each Such That Clause. Then, it will check if the argument type is in the valid domain of itself when adding the new SuchThatClause to the clauses. For example, the valid domain for the left argument of Affects relationship is {STMT, ASSIGN, WILDCARD, STMTNUMBER}. When the parser realises that the left argument of the Such That clause in the query is of type VARIABLE, but it does not exist in the valid domain. Therefore, the query will evaluate to semantically invalid.

Checking for the type matching of With clauses

For the four types of attributes we have in SPA, there is a value domain for them.

The mapping of the domain is as following:

value, stmt#	INTEGER
procName, varName	IDENT

When the parser parses a with clause, it will define the value domain of the left and right

argument. For example, if the argument is an attribute reference, it will get the value domain of it by referring to the map. If the argument is an INTEGER or IDENT, it will be defined accordingly. Eventually, the parser will check if the value domains of the left and right argument match. If they do not, the query will evaluate to semantically invalid.

Handling of the semantic invalidity:

The parser will set the attribute `is_semantically_valid` in the query object as `FALSE` if it detects a semantic error. Since a syntactic error takes precedence over a semantic error, the parser will continue parsing to check for the syntactically validity of the query. If a syntactic error is detected later, the handling for syntactic error will come first.

Interaction with Query Evaluator

After the parser has parsed the query text, a query object will be created and it will be passed to the query evaluator for evaluation. Then, the Query Evaluator can use the APIs provided by Query Parser in the Query class to extract the clauses and synonyms for evaluation.

When parsing the clauses, the parser uses the constructors provided by Query Evaluator to create Clause objects according to the types of the clauses.

1.4.1 Design decisions

Design consideration #1: One-pass Parsing (Parsing+Validation+Query Object Creation)

Two different variations of parsing strategy have been considered, which is one-pass parsing that we have decided to use and two-pass parsing (Parsing+Syntactic Validation -> Semantic Validation + Query Object Creation).

Criteria	One-pass Parsing	Two-pass Parsing
Traversal of the query	The parser needs to traverse the query text once in total, in which the validation of the query and the query object creation will be done.	The parser needs to traverse the query text twice. The first time is to validate if the query text is syntactically valid. The second traversal is to check for the semantic validity of the query and create the query object.
Expected time for parsing	If the query is syntactically invalid and the PQL	If the query is syntactically invalid and the PQL

	grammar-violating part is towards the end of the query, this parser will take longer as the populations of the synonyms and clauses in the query object takes up the additional time. However, if the query is syntactically valid, the parser will be faster because it does not need to traverse the query text before to check for the validity.	grammar-violating part is towards the end of the query, this parsing strategy will end faster. However, if the query is syntactically valid, the parser will need more time to complete the parsing compared to one-pass parsing.

After comparing the two strategies, we have decided that one-pass parsing is the better option as the overhead for the query object creation is not that significant. Also, the parsing will stop when the parser encounters the part that violates the PQL grammar. This will also help shorten the time needed for parsing as the parsing of the remaining part of the query will be wasteful. Moreover, since most of the queries are syntactically valid (mentioned in wiki), the first approach would be more suitable here.

Design consideration #2: Using a set of valid entities for each argument of the Such That clauses and With clauses

Instead of using switch cases to check for the semantic validity for each argument of all Such That clauses, we go with using a map that maps each type of relationship to the domain of valid entities of its arguments. This will introduce better extendability because we just need to modify the domain set for each relationship if we have a change in the valid domain. Also, we can just add a new mapping of relationship and its valid domain for the arguments if we introduce a new relationship. This design adheres to the Open-Closed Principle, which means the code can be extended without modifying the code structure tremendously. Besides that, the readability of the code will improve as we do not need to read a long list of branches for validating the arguments of each relationship.

For With clauses, we just need to add the mapping of attribute to its domain when a new attribute is introduced. The domain matching logic for the left and right hand side of the with clause will still be working even if we add a new attribute type.

Design consideration #3: Setting a flag for semantically invalid query in the query object instead of throwing an exception

Syntactic validity takes precedence over semantic validity. Hence, when the query is found to be semantically invalid, the parser should continue parsing until the end of the query text. Since the parsing will not stop, we have decided to set a flag for semantically invalidity in the query object when we encounter the part that violates the semantic validity. Since there is no need to stop the parsing, we have decided not to use an exception to handle this.

1.4.2 Abstract API

The most important APIs provided by Query Parser are listed below.

APIs	1-2 line description
LIST_ATTRREF GetAttrRef()	Return a list attribute references for the Select result clause
LIST_PTR_CLAUSE GetClauses()	Return a list of pointers to the clause objects
BOOLEAN GetBoolean()	Return true if the query is selecting BOOLEAN, false otherwise.
BOOLEAN IsSemanticallyValid()	Return true if the query is semantically valid, false otherwise.

The full list of APIs provided by Query Parser are in the appendix (Section 8.3 Query Parser Abstract API).

1.5 Query Evaluator

Query Evaluator

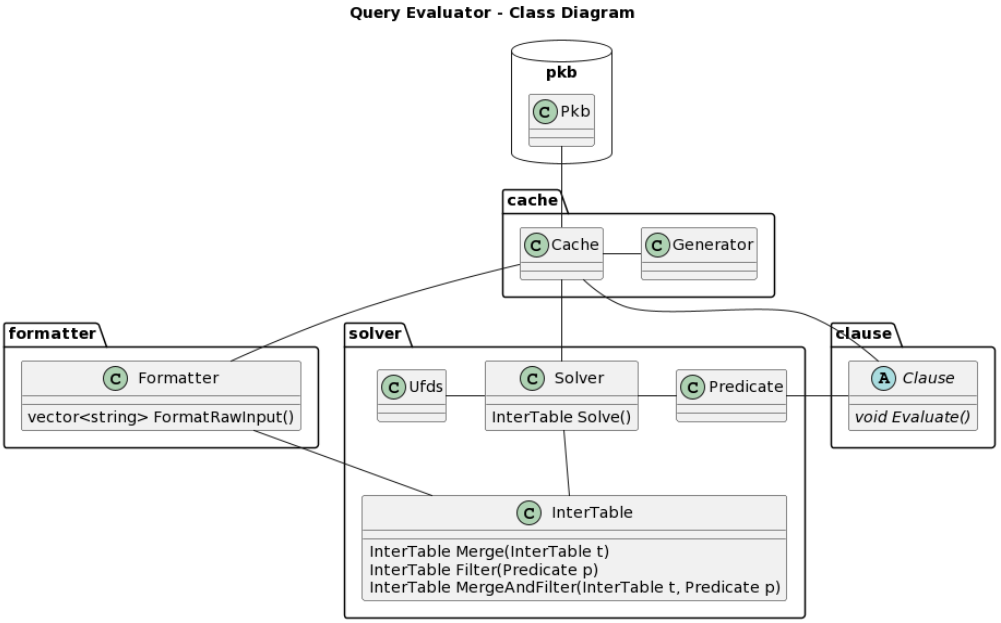

Diagram 1.5.1 Class diagram for Query Evaluator

The classes of Query Evaluator can be group into four major packages:

1. **cache** : Act as a facade class between PKB and Query Evaluator, computes and stores the complex relationship
2. **clause** : Represent a clause and handle the logic of evaluating a clause
3. **solver** : Come up with the final table that satisfy all clauses
4. **formatter** : Format a raw input table into a vector of string

Query Evaluation Strategies and Optimisation

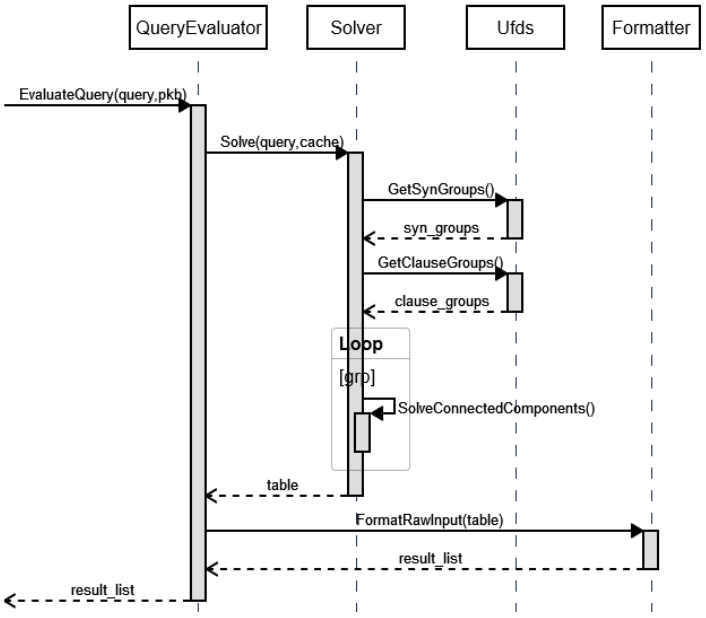

Diagram 1.5.2 Sequence diagram for EvaluateQuery function

The EvaluateQuery function mainly consists of two steps: calls Solve in Solver to get the final table that satisfies all clauses and calls Formatter to format the final table into a list of strings. A cache object will be instantiated and will be used in the two steps mentioned.

Cache and Generator

Cache acts as a facade class between Query Evaluator and PKB and also responsible to store the `Next*/Affects/Affects*` relationships after they have been computed for efficient retrieval later. `Generator` class will be used to compute all the complex relationships.

1.Calling Solve in Solver to get the final table

As an optimization, Solver will divide the clauses into multiple groups using Ufds. Clauses in the same group have connected synonyms. After that, Solver will solve each group separately by calling the `SolveConnectedComponent` function. We used brackets to indicate that the steps belong to the `SolveConnectedComponent` function.

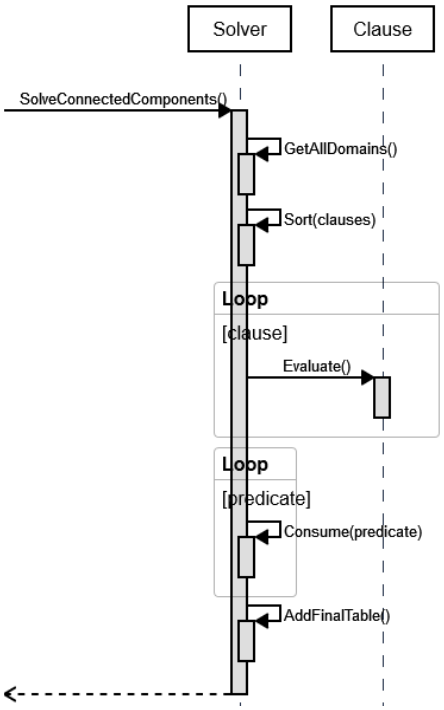

Diagram 1.5.3 Sequence diagram for SolveConnectedComponent function

(1.1) Get initial domains for used synonym in groups

In each group, Solver will first get the domain of all used synonyms in the current group. A synonym is considered used if it is involved in any clauses. All synonyms are represented as integers including procedures and variables.

(1.2) Sort the clauses within group

We prioritize clauses with only one synonym involved. If the number of synonyms is the same, the clauses are sorted by type. The clauses are prioritizes in the following way (highest priority to lowest priority) based on how restrictive the clauses are:

- With clause
- Follows, Parent, Calls, ModifiesS, Next, pattern clause
- Follows*, Parent*, Calls*, UsesS, ModifiesP, UsesP, NextT
- Affects, Affects*

(1.3) Evaluate all clauses in the group

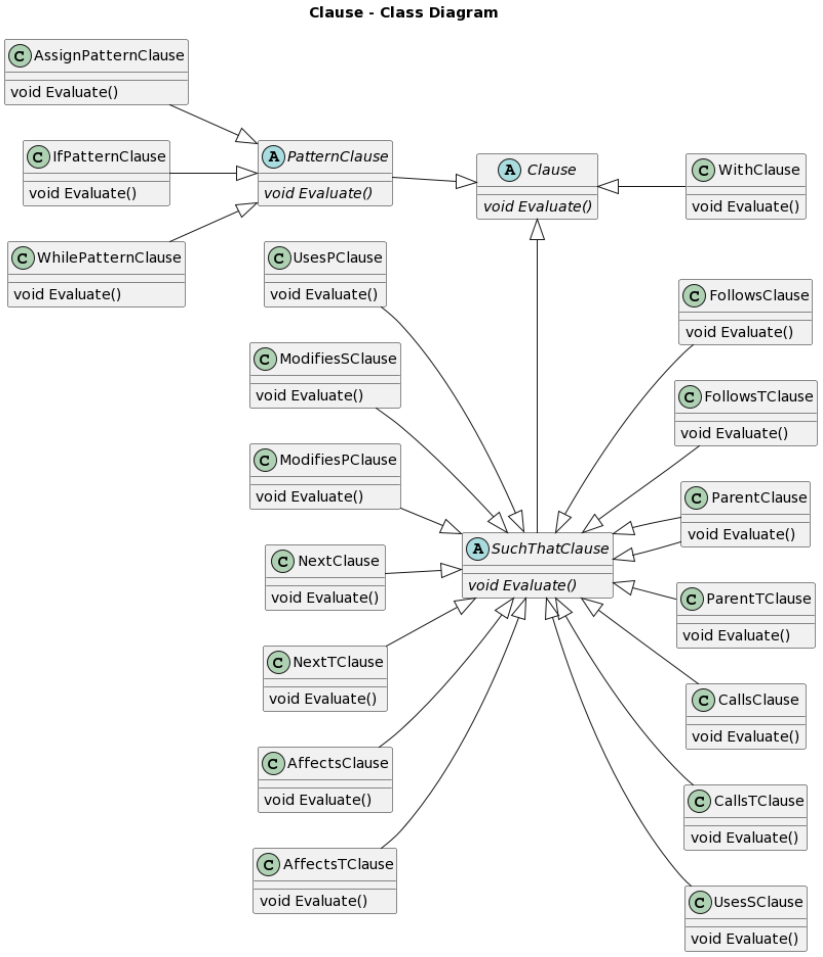

Diagram 1.5.4 Class diagram for package clause

For clauses with 0 synonyms involved, they will always be placed in the first group. For example, Parent*(3,7), Modifies(5, “x”), Solver will inquire the cache whether the relationship exists. If the relationship does not exist, a **`FalseRelationException`** will be thrown.

Moreover, for clauses with 1 synonym involved, e.g. Parent(s, 3), Uses(3, v), query evaluator will get the list of entities that satisfy the relationship and the synonym’s domain will be trimmed by intersecting the domain with the list returned by PKB. If the domain is empty after intersection, an **`EmptyDomainException`** will be thrown. Query evaluator allows early termination if the boolean value returned by PKB is false or a domain is empty to speed up the query evaluation. Also, by trimming the domain of the synonym, this will reduce the number of rows of InterTable which could increase the query evaluation speed.

Lastly, clauses with 2 synonyms involved are considered non-trivial clauses and a Predicate object will be created and added to the predicate list. A Predicate object consists of two synonyms and a list of allowed value pairs. For instance, Follows(s, s1) will create a Predicate object where `first_= s, second_= s1` and `allowed_pairs_= [<1,2>, <2,3>, ...]`. The predicate lists will then be used to find synonym values that satisfy all predicates in the current group.

(1.4) Consume each predicate

InterTable

Solver uses InterTable to store the intermediate results and apply merging and filtering to get the final result. InterTable consists of two attributes, which are `headers_` and `rows_`. `headers_` stores the name of the synonym in each column whereas `rows_` is essentially `vector<row>`, where the number of columns in each row will be the same. The three functions below are the most important function in InterTable:

APIs	Description
InterTable Merge(InterTable t)	Apply a cross product between rows_ in the current InterTable and rows_ in t. The time complexity will be $O(n^2)$.
InterTable Filter(Predicate p)	Filter out rows that the value of synonyms in Predicate is not inside the allowed pair.
InterTable MergeAndFilter(InterTable t, Predicate p)	Apply a Merge and Filter to the current InterTable. Solver will try to use this function (if possible) because it would consume less memory space and is faster than calling Merge and Filter subsequently.

Solver will first create an InterTable for every synonym used in the current group. Next, Solver will consume each Predicate and apply MergeAndFilter if the two synonyms in Predicate are in different InterTable. Otherwise, Filter will be applied if both synonyms in Predicate are in the same table. Every synonym involved in a Predicate must be guaranteed to be inside one of the InterTable because Solver created an InterTable for every synonym used by definition.

If any result table after applying Filter or MergeAndFilter is empty, an `EmptyTableException` will be thrown. This serves as an early termination condition and will optimize the query evaluation.

After all predicates are consumed, only one table will remain (except for the first group where there are no synonyms involved) since all synonyms are connected in each group and

all tables will eventually be merged together. Next, Solver will remove the synonyms that are not returned, which are essentially the columns of the table and apply a deduplication to speed up the merging operation later. The result table will then be added to a list for a final merge after all connected components are solved.

2.Format the final table into vector of string

Given the final table, the formatter will get the string representation of each selected element and compile them into a single vector of string.

Example

Here we show one example of how query evaluation is performed. We will be referring to the source program in Section 1.1.

First example

```
assign a; stmt s; if ifs; variable v;
Select <a, ifs> such that Follows(s, a) pattern a (_, _"x"_) and ifs (v, _, _) with 1 == 1
```

1. Group the clauses

The clauses will be divided into the following groups:

- 1 == 1
- Follows(s, a), a (_ , _"x"_)
- ifs (v, _, _)

2.1 Solve group 1

The with clause will be evaluated to true and hence the Solver will continue.

2.2 Solve group 2

2.2.1 Get domain of all used synonyms

s:{1, 2, 3, 4, 5, 6, 7,8,9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28}

a : {1, 6, 7, 8, 9, 12, 13, 16, 17, 22, 23, 24, 28}

2.2.2 Sort the clauses in group 2

pattern a (_, _"x"_) will be placed before Follows(s, a)

2.2.2 Evaluate clauses in group 2

For the pattern clause, the list retrieved from PKB will be [6, 9, 12]. Hence, after applying an intersection, we get the following domain:

s:{1, 2, 3, 4, 5, 6, 7,8,9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28}

a : {6, 9, 12}

For Follows(s,a) clause, a Predicate object is created and added to predicates.

predicates : [{“first_” : s, “second_” : a, “allowed_pairs_” : {<1, 2>, <2, 3>, <3, 11>, <4, 9>, <7, 8>, <9, 10>, <11, 14>, ……}}]

An InterTable is created for each synonym used :

s
1
2
3
…
26
27
28

a
6
9
12

2.2.3 Consume predicates

Apply MergeAndFilter to consume the predicate:

s	a
4	9

2.2.4 Remove columns that are not returned

The table added to the list will be:

a
9

2.3 Solve group 3

Similarly, for group 3, the following table will be added:

ifs
4
11
18
19

3. Merge all tables from each group

Finally, merge all tables together by applying a cross product:

a	ifs
9	4
9	11
9	18
9	19

Interaction with PKB

In the actual implementation, there will be no direct association between the clause classes and PKB. All interaction will be done through the cache class. The cache class will have the same function declaration as the APIs provided by PKB. The discussion below will ignore the existence of cache class since we are interested in the APIs provided by PKB and how are used in Query Evaluator.

Interaction between Such That Clause and PKB

Generally, for each such that clause, PKB will provide 5 different APIs.

bool IsRelationshipHolds(ClauseType type, Entity a, Entity b)
Return true if Relation(a, b) of type exists. Otherwise, return false.

List<Entity> GetRelSecondArgument(ClauseType type, Entity a)
Return all entities x such that Relation(a, x) of type holds true.

List<Entity> GetRelFirstArgument(ClauseType type, Entity a)
Return all entities x such that Relation(x, a) of type holds true.

List<pair<Entity, Entity>> GetRelArgumentPairs(ClauseType type)
Return all entities pair <x, y> such that Relation(x, y) of type holds true.

bool IsRelationshipExists(ClauseType type)
Return true if there exists at least one pair of entities <x,y> such that Relation(x, y) of type holds true. Otherwise, return false.

The argument type is an enum that uniquely defines a such that clause. For example, the clause Follows(5, 6) will call `IsRelationshipHolds(Follows, 5, 6)` provided by PKB.

Interaction between Pattern Clause and PKB

For assign pattern clauses, PKB provides two APIs to deal with the expression matching in the RHS. For LHS, it is the same as doing a ModifiesS such that clause and hence no extra API is needed.

List<Assign> GetStmtsWithExactPattern(string expression)
Return all assign entity x such that its RHS is exactly the same as expression.

List<Assign> GetAllStmtsWithPattern(string expression)
Return all assign entity x such that its RHS contains expression as subtree.

For if and while pattern clause, PKB provides three APIs:

List<VarEntity> GetAllPatternVariablesInStmt(ContainterEntity a, PatternType type)
Return all variable entity x such that x is used in the condition of a statement. The type will decide whether it is an if or while statement.

List<ContainerEntity> GetAllStmtsWithPatternVariable(VarEntity a, PatternType type)
Return all container entity x such that x uses a in its condition. The type will decide whether it is an if or while statement.

List<pair<ContainerEntity, VarEntity> GetContainerStmtVarPair(PatternType type)
Return all pair <x, y> such that y is used in the condition of x. The type will decide whether it is an if or while statement.

Interaction between With Clause/Formatter and PKB

With Clauses and Formatter require the same APIs provided by PKB as both of them need to handle attributes of synonyms. The APIs provided are mainly to convert variable/procedure strings to the index stored in PKB in both directions and getting the variable involved in read/print/call statement.

Design Pattern

Command Pattern

Referring to diagram 1.4.5, one design pattern used in Query Evaluator is the command pattern. For every clause, the Query Evaluator can evaluate the clause without knowing the type of the clause. This will allow the query evaluator to follow the open-closed principle as we can easily add new clauses by extending from the Clause parent class without having other existing clause classes modified.

Facade Pattern

The cache class acts as a facade class between Query Evaluator and PKB. This will significantly reduce the coupling between classes in Query Evaluator and PKB.

Data-driven programming

Moreover, for 'such that' Clause, data-driven programming is used in the Evaluate function so that the code is more flexible. The Evaluate function will choose a function pointer based on the type of left and right argument.

Left argument \ Right argument	**Wildcard**	**Entity**	**Synonym**
Wildcard	EvaluateWildWild	EvaluateWildEnt	EvaluateWildSyn
Entity	EvaluateEntWild	EvaluateEntEnt	EvaluateEntSyn
Synonym	EvaluateSynWild	EvaluateSynEnt	EvaluateSynSyn

By implementing this way, we break the Evaluate function into 9 small functions that deal with a specific combination of arguments. This follows the DRY principle as we will not repeat the codes for such that clauses with slightly different evaluation logic. For example, only Modifies and Uses clauses cannot have wildcard as the left argument.

Furthermore, once the function is decided, it is clear which 5 APIs provided by PKB are to be used. For example, if the function is `EvaluateEntSyn` the API needed to be called would then be `GetRelSecondArgument` so that we can trimmed the domain of the synonym involved.

For summary, using data-driven programming in the Evaluate function will make the code more flexible as we can easily add new types of such that clauses without adding many lines of code. It will be easy to test and debug since each function handles a specific combination

of arguments. The Evaluate function for pattern clause and with clause are written in a similar fashion.

1.5.1 Design decisions

Design Decision #1: Using vector of vector to store the values in InterTable

Another alternative is to use hashmap to store the values in InterTable. The consideration taken here is the time efficiency. InterTable will require merging which will cause the table size to be $O(n^2)$ of the original table size. When the table size exceeds the current hashmap capacity, all elements in hashmap will need to be rehash and this will cause a delay in query evaluation. Hence, we decided to go with using vector of vector to store the values in InterTable.

Design Decision #2: Using merging and filtering to reach the final result

We can view the problem of finding all valid values such that they satisfy all clauses as a Constraint Satisfaction Problem (CSP) <X, D, C> where X is the set of all used synonyms, D is the domain of all used synonyms and C is the set of constraints which are the clauses in query evaluation. Backtracking is a popular method to solve CSP in the Artificial Intelligence field.

However, we are not familiar with backtracking, which will require more time and manpower to implement. Also, due to the complexity of backtracking, it is more bug-prone and our team has decided to emphasize more on correctness than query evaluation speed.

1.5.2 Abstract API

The most important APIs provided by Query Evaluator are listed below.

APIs	**1-2 line description**
LIST_STRING EvaluateQuery(QUERY query, PKB pkb)	Return a list of strings that satisfy all the clauses in the query.

The full list of APIs provided by Query Evaluator are in the appendix (Section 8.4 Query Evaluator Abstract API).

2 Testing

2.1 Unit testing

2.1.1 Unit testing for SP

Unit testing for Source Processor is primarily focused on tokenization and validation.

Tokenizer

We created an input string for each test case and passed it to the tokenize function. For every list of tokens returned by the Tokenizer, we then checked the correctness of each token in the list using assertions.

Validation

We provided various valid and syntactically/semantically invalid SIMPLE source programs which are used as input for positive and negative test cases. For valid test cases, each source program is parsed by the parser from start to finish (i.e. create parser object, call validate() function and populate() function of parser object). The assertion REQUIRE_NOTHROW() was used at each step to ensure that no exceptions are being thrown by the parser.

For invalid test cases, the parser object is first created then the validate() function is called. Since the populate() function will not be reached if there is an InvalidSyntaxException or InvalidSemanticException when running the actual SPA, it is irrelevant for these test cases and hence will not be called. The assertion REQUIRE_THROWS_AS() was used to check if the function call validate() will throw InvalidSemanticException for semantically invalid test cases. The same assertion REQUIRE_THROWS_AS() is also used to check if the program will throw an InvalidSyntaxException when creating new Parser objects for syntactically invalid test cases.

Control-Flow Graph

There is also a set of test cases for the testing of CFG, to check that the CFG generated for each test case is the same as the expected CFG. This is done by implementing an equal function for the CFG to compare the cfg generated using the GenerateCfg() function against the expected Cfg, which is built manually for each test case.

The different source programs used in our unit testing aim to cover a wide range of possibilities for tokenizing and validating, since the actual input SIMPLE source program is unpredictable. Our first batch of test cases used source programs which only had read/print/assign statements, the second batch included if/while statements and the third batch had 2 or more levels of nesting. We started off with the first batch of test cases and only proceeded on after all the test cases in the previous batch passed.

2.1.2 Unit testing for PKB

Generally, the unit tests for PKB focuses on the private implementation as well as the public APIs. For the first sample unit test provided, it focuses on the private implementation of the hash tables storing the relationship information. The test seeks to ensure that the underlying functions of the custom table are working as intended. That is it manages to add the key-value pair into the table, gets the value by key successfully, throws an exception if the key cannot be found and so on. While there are many possible table types that can be created using the table template, we have decided to focus only on the table with integer type for both the key and value. An example of the test suite used in the SPA is provided below.

Testing Follows Table

- Step 1: Initialize a FollowsTable
- Step 2: Test API for getting key list when table is empty
- Step 3: Add key-value pairs and check if insertion is successful by checking the insertion status (true if successful) and the element in the table. Test if the API can throw custom exception if key already exists in the table
- Step 4: Test the API for getting value from table by key; Check if the API throw custom exception if key does not exist in the table when retrieving value by key
- Step 5: Test the API to get key list from table
- Step 6: Test the API to get key-value list from table
- Step 7: Test the API to update value in table

As mentioned earlier in Section 1.3.1, the PKB will also be populating the reverse relationships for certain relationships such as Follows. As such, for the second sample test suite provided below, it will focus on testing that the reverse relationships are populated successfully behind the scenes. This helps to ensure that the private implementation for this function in the PKB is working as intended. To verify that the reverse relationships were indeed populated correctly by the PKB, the public APIs that the PKB exposes will be used to test.

Test populating Follows and FollowsBefore Table

- Step 1: Initialize the PKB instance
- Step 2: Insert the Follows relationship pairs
- Step 3: Verify if the reverse relationship pairs are correctly stored in the FollowsBefore table

2.1.3 Unit testing for PQL

Query Parser

The unit testing for Query Parser will be on the parse function of the Parser class. Different query examples are provided to test the correctness of the parse function – both valid and invalid.

For invalid queries, we will separate the test case into different sections that take care of different cases of invalid queries.

The sections for invalid queries:

1. With no Select clause (just declarations)
2. With wrong spelling of keywords
3. Query contains synonyms that are not declared
4. The first argument of Uses and Modifies is a wildcard
5. The second argument of Uses and Modifies is not a variable
6. Assign pattern clause such that the expression is not valid
7. Select result clause with invalid attributes
8. Repeated synonym names declared
9. Syntactically invalid arguments in Follows/Follows*
10. Syntactically invalid arguments in Parent/Parent*
11. Syntactically invalid second arguments in Uses
12. Syntactically invalid second arguments in Modifies
13. Syntactically invalid arguments in Calls/Calls*
14. Syntactically invalid arguments in Next/Next*
15. Syntactically invalid arguments in Affects/Affects*
16. Syntactically invalid first argument in pattern clauses
17. Syntactically invalid pattern synonym
18. Empty Select tuple

For valid queries, we will separate the test case into different sections as well.

The sections for valid queries:

1. With Select clause only
2. With Select and such that clause
3. With Select and pattern clause
4. With Select, such that and pattern clause
5. With Select and pattern clause with expression
6. With Select result clause with synonym and attribute
7. With Select tuple and attribute
8. With Select and with clause
9. Implicit and
10. Explicit and

For each section, a few queries will be created to make sure that the test case is extensive enough.

Query Evaluator

The unit testing for Query Evaluator mainly focuses on InterTable as it does not interact with other subcomponents. For InterTable, the test ensures the correctness of Deduplication, Merge, MergeAndFilter and Filter.

Purpose:

Ensure the correctness of MergeAndFilter in InterTable

Requirements:

InterTable does not depend on other classes and hence can be tested individually.

Sample unit test case:

Here we use tables to represent InterTable as it is easier to visualize.

predicate = {“first_” : s, “second_” : v, “allowed_pairs_” : {<2, “a”>, <8, “c”>, <4, “b”>, <8, “d”>, <2, “c”>, <2, “d”>}}

To test: `t.MergeAndFilter(t1, predicate)`

t

s
2
4
6
8

t1

v
a
b
c
d

Expected test result:

s	v
2	a
2	c
2	d
4	b
8	c
8	d

2.2 Integration testing

For integration testing between the SP and PKB, the test cases focused mainly on the population of entities and relationships by the parser and design extractor. This is done using 2 independent test cases, which are split into the population of entities and basic relationships which is done by the parser and the population of nested relationships which is done by the Design Extractor.

Integration testing between SP parser and PKB

First, testing was done on the population of entities and basic relationships of the SIMPLE source program by the parser into the PKB. Each test case will take in a valid SIMPLE source program and create a PKB instance, which is used to create a Parser object. Then, the Populate function of the Parser will be called for the PKB to be populated. After the PKB has been populated, the contents of all the entity sets (i.e. procedure/variable/constant set), entity tables (i.e. read/print/assign/if/while/call tables) and relationship tables (i.e. Follows/Parent/Uses/Modifies/Calls/Next tables) will be checked for correctness using assertions and the APIs provided by the PKB team.

Example test case

Purpose:

Test the population of Parent relationship

Requirements:

The SIMPLE source program passed into the Paser should be syntactically and semantically valid, else the test case will throw an unhandled exception which will cause it to stop. The test case will also require a working API to retrieve the contents inside the Parent Table, which is provided by the PKB team.

Expected Results:

Using the SIMPLE source program written in Section 1.1, more specifically the teamReport procedure as an example, the Parent Table should contain the values listed in the table below.

Key	Values
3	{4, 9, 10}
4	{5, 7, 8}
5	{6}
11	{12, 13}
18	{19, 22, 23}
19	{20, 21}

Nested relationships such as Parent* and Follows* will not be tested during this step of integration testing for the parser.

Integration testing between SP Design Extractor and PKB

After the parser has added the basic relationships to the PKB, the design extractor will then be called to populate the nested relationships such as Follows*, Parent* and Calls* and these cannot be done until the end. To do this, the design extractor will require certain tables from the PKB to come up with the nested relationships. Following the previous example given in the previous paragraph, after the Parent table has been populated, the design extractor will populate the PKB with the nested Parent relationships listed below.

Key	Values
3	{4, 5, 6, 7, 8, 9, 10}
4	{5, 6, 7, 8}
5	{6}
11	{12, 13}
18	{19, 20, 21, 22, 23}
19	{20, 21}

Population of other nested relationships such as Follows*, Calls* and ModifiesP etc. will also be done in a similar manner.

As such, in testing the integration of the Design Extractor with the PKB, the PKB was first populated with the basic Follows and Parent relationships. Subsequently, the population method in the Design Extractor was called to populate the nested relationships into the PKB. Finally, to test that the Design Extractor was indeed getting the relevant tables from the PKB and using the information to populate the nested relationships, the public APIs provided by the PKB in querying the tables were then called to ensure that the nested relationships were indeed populated correctly.

For example, when we test for the population of Follows* relationship through the design extractor, we will first initialize the database and add the direct statement number pairs that satisfy the Follows relationship to the FollowsTable. We then call the populator in the Design extract to populate the Follows* relationship into the respective tables. Afterwards, we

verify the correct key-value pairs for Follows* relationships are stored in the tables and no duplicate is found.

Example test case

Purpose:

Test the population was successful for the Follow* relationship

Expected test results following example from Section 1.1:

```
Follows*(1,2) => true
Follows*(1,3) => true
Follows*(4,9) => true
Follows*(4,10) => true
Follows*(11,14) => true
Follows*(9,10) => true
Follows*(11,12) => false
```

Integration testing between Query Evaluator and PKB

For integration testing between query evaluator and PKB, we mainly focus on the correctness of evaluating each Clause class and pattern clause.

For each Clause class, we test for all possible combinations of arguments. For example, for FollowsClause with no synonym involved, we test for `Follows(_, _), Follows(1, _), Follows(_, 3), Follows(4, 5)`. In each possible combination, we tested for positive cases and negative cases as well. We ensure that the domain of the synonym is accurately trimmed and the Predicate object is created correctly. Also, ensure that negative test cases are correctly handled.

Example test case

Purpose:

Ensures the predicate created by the clause is correct

Requirements:

Since the evaluation of the Clause class requires a Pkb object as their input, we need to manually populate the relationship ourselves to simulate the testing environment. Also, the SIMPLE source program and query used are also valid.

Sample test example from Section 1.1:

```
assign a; variable v; Select a such that Uses(a, v) pattern a (v, _)
```

We mainly focus on testing the UsesSClause class.

Expected test results:

The size of predicates (a list containing all predicates generated by each clause) is increased by 1. Also, the last element is equal to {“first_” : a, “second_” : v, “allowed_pairs_” : {<6, “x”>, <6, “y”>, <7, “tuesday”>, <7, “count”>, <8, “friday”>, <8, “count”>, <9, “x”>, <12, “x”>, <12, “y”>, <13, “monday”> , <13, “count”>, ……}}.

2.3 System testing

For system testing, we created several SIMPLE source programs and queries for each program to ensure that the SPA returns results as expected. Our design for both the SIMPLE source code and PQL queries is explained below.

Designing the SIMPLE source code

When designing the SIMPLE source code, we wanted it to be complex enough to ensure that our SPA would be able to handle parsing, population of PKB and evaluating queries adequately. For instance, we included

- Variable names that are the same as procedure name and terminals
- Different type and amount of whitespaces
- Nested code (e.g. While/If statements within While/If statements)
- Different types of operators and brackets

From Iteration 2 onwards, we also included multiple procedures in the source codes. For Iteration 3, we created longer source codes with over 70 statements.

Invalid SIMPLE source programs were created to ensure that recursive and cyclic procedures in the program will correctly cause the SPA to stop evaluating.

Designing PQL queries

When designing PQL queries for each SIMPLE source program, we followed equivalence partitioning so that many different partitions can be tested without creating too many queries. Some partitions include the number of clauses (from 0 to multiple), use of synonyms and/or constants in parameters, type of relationship (eg. Modifies, Uses), number of common synonyms and type of return value for the query. We also made sure to test nested relationships (eg Follows*/Parent*). When writing queries, we tested combinations of ‘such that’ and ‘pattern’ clauses, as well as the use of explicit ‘and’ operators, to ensure that our query evaluator was able to handle all types of queries. Besides positive test cases, we also wrote semantically and syntactically invalid queries to check that the SPA does not return any answer.

In the following sections, we will elaborate on the System Testing for each iteration.

2.3.1 System Testing for Iteration 1

For iteration 1, we used 3 sets of source programs and their corresponding queries to perform system testing. Below is one of the test cases we used, which consists of a SIMPLE source program and a set of PQL queries for that source program.

Sample source program

```
procedure modify {

while (   awesome >                    20                          )


{x = modify * 2;

read y;

y = modify * 2;

if (ready == 10) then {

x = z + (x + i) + z + c * (a - b / e) * (a % a);

print x;

} else {

a = modifies * 2 + x + i;

y = hello; }

b = modifies * 2; print b;

read hello;

} }
```

Examples of PQL queries for the above SIMPLE source program:

Partition	Test case
No clauses	1 - Return all read statements read r; Select r 3, 12 5000
Multiple clauses	2 - Select all the assignments which Modifies v and Uses v assign a; variable v; Select a such that Uses(a, v) pattern a(v, _) 6 5000
Use of synonyms only	3 - Select all assignments that uses a variable assign a; variable v; Select a such that Uses(a, v) 2, 4, 6, 8, 9, 10 5000
Use of synonym and constant	4 - Select all the statements such that 1 is a transitive parent of it stmt s; Select s such that Parent*(1, s) 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 5000
No common synonyms between clauses	5 - Select all the statements such that it follows a if loop assign a; if ifs; stmt s; variable v; Select a such that Follows(ifs, s) pattern a(v, “x”) 5000
1 common synonym between clauses	6 - Select all the assignments such that it has a subtree pattern and that the variable is used by a if statement assign a; if ifs; variable v; Select a such that Uses(ifs, v) pattern a(v, _"x"_) 6, 8 5000
Invalid query	7 – Invalid query, Select all the statements that modifies a variable variable v; Select s such that Modifies(s, v) 5000

2.3.2 System Testing for Iteration 2

For iteration 2, we implemented the following subset of Advanced SPA requirements:

1. Boolean Return
2. Tuple Return
3. Multi-clause
4. Explicit and
5. With Clause
6. Pattern Sub-expression for assignment
7. Calls, Calls*

We performed system testing for each of these features to ensure that all components are able to support them. One set of queries is created for each feature to test them individually, and the queries are partitioned as follows:

- Number of clauses (0, 1, 2+)
- Use of synonyms and/or constants
- Basic relationships & nested relationships
- Number of common synonyms between clauses (0, 1, 2+)
- Different number and combination of ‘such that’ and ‘pattern’ clauses
- Syntactically invalid queries (SPA should not return an answer)
- Semantically invalid queries (SPA should return FALSE for BOOLEAN return instead of empty answer)

For features 1-5, we used the following SIMPLE source program for system testing:

```
procedure procedure {
      read read;
      if ((read == 0) || (print ==0) ) then {
            x  = 10000;
            y = x * z * t + (5 + 4);
            print procedure;
      } else{
            while (while!= 0) {
                  print print;
                  while = while + 1;
                  if ((while == 0) &&(z > 9)) then{
                        print p;
                        p = 289 * 444 + (f * cenX);
                        call k;
                  } else {
                        read story;
                  }
            }
            xxxx = xy + (99 * (((4) + y) * 9));
            call Procedure;
      }
}
procedure k {
      call Procedure;
      print = 2 + xxxx;
      read abc;
}
procedure Procedure { print z; }
```

Examples of PQL queries for the above SIMPLE source program:

Partition	Test case
No clauses (Boolean return)	1 - Synonym that is named BOOLEAN print BOOLEAN; Select BOOLEAN 5, 7, 10 5000
2 clauses (With clause)	2 - Valid synonym return with with clause stmt s; constant c; Select s such that Uses(s, "while") with s.stmt# = c.value 9 5000
2 clauses (Tuple return, Multi-clause)	3 - Valid tuple return with multiple assign pattern clauses assign a, a1; Select <a, a1> pattern a ("p", _"cenX"_) pattern a1 ("xxxx", _"xy"_) 11 14 5000
2 clauses (Tuple return, Explicit and)	4 - Valid tuple return with pattern and explicit and. Similar to 15 assign a, a1; Select <a, a1> pattern a ("p", _"cenX"_) and a1 ("xxxx", _"xy"_) 11 14 5000
3 clauses (Multi-clause)	5 - Valid boolean return with multiple such that and assign pattern clauses assign a; if ifs1, ifs2; while w; Select BOOLEAN such that Parent(ifs1, a) pattern a ("x", _) such that Parent(ifs1, ifs2) TRUE 5000
Syntactically invalid query	6 - Tuple return for syntactically invalid query stmt s; Select <s.stmt> 5000
Semantically invalid query	7 - Boolean return for semantically invalid query stmt s; Select BOOLEAN such that Modifies(r, "read") FALSE 5000

To test for feature 6 (Pattern Sub-expression for assignment), we used the following SIMPLE program which involves complex assignment patterns:

```
procedure patterns {
      x = x * y + f + a + assign * g;
      y = (f * q * e - s * 9) * (g + (10 + a) * y);
      z = 100 * 9 + a * d / r + (a * (f + (w / t)) - x - 100);
      q = a * f - 0 + g * i / t + (f + (w / t));
      q = 1;
}
```

For each query in this test case, there are only pattern clauses (no such that/with clauses) as the main focus is to test if SPA is able to parse all the pattern substrings and PQL is able to identify them correctly.

To test for feature 7 (Calls/Calls*), we used the following SIMPLE program which has many procedures and involves complicated call flows:

```
procedure p1 { p1 = p1; }

procedure p2 { p2 = p2; call p1; call p1; }

procedure p3 {
      p3 = p3;
      while(1==1) { call p2; }
      call p1;
      call p2; }

procedure p4 {
      p4 = p4;
      if (1==1) then {
            call p3;
      } else {
            call p5;
            call p3;
      }
      call p3; }

procedure p5 { p5 = p5; }

procedure p6 { p6 = p6; call p1; call p2; call p3; call p4; call p5;
}
```

For each query in this test case, only such that clauses with Calls/Calls* are used as the main focus is to test if SPA can parse multiple procedures with complex call flows and PQL can correctly identify the Calls and Calls* relationship.

2.3.3 System Testing for Iteration 3

For iteration 3, we finished implementing the following requirements to complete the Advanced SPA requirements:

1. If/While patterns
2. Next, Next*
3. Affects, Affects*

To test for the correctness of If/While patterns, a SIMPLE program was created with if and while statements. Similar partitions for test cases as above were created to check that both valid and invalid tests return results as expected.

Examples of PQL queries:

Partition	Test case
Multiclause (Boolean return, explicit and)	1 - Multiple pattern with explicit and if ifs; while w; Select BOOLEAN pattern w ("while", _) and ifs ("while", _, _) such that Parent(w, ifs) TRUE 5000
Multiclause (Tuple return, explicit and)	2 - Multiple pattern clauses if ifs; assign a; while w; variable v; Select <ifs, a> such that Parent(ifs, a) pattern ifs(_ , _ , _) and w (v, _) and a (_ , _) 2 3, 2 4, 2 14, 9 11, 17 18 5000
Syntactically invalid query	3 - Syntactically invalid number of arguments for while pattern while w; Select BOOLEAN pattern w ("while", _, _, _) 5000
Semantically invalid query	4 - Semantically invalid number of arguments for if pattern while w; Select BOOLEAN pattern w ("while", _, _) FALSE 5000

To test for the correctness of Next/Next* and Affects/Affects*, we created a SIMPLE program which makes use of assignment statements and multiple procedure calls. There is also nested code with up to 6 nested container statements.

We created basic queries and manually checked that the answers were correct. Examples of PQL queries created:

Relationship	Test case
Next	stmt s1, s2; Select <s1, s2> such that Next(s1, s2) 75 76, 69 70, 70 71, 70 74, 71 72, 72 73, 73 70, 31 32, 32 33, 33 34, 34 64, 34 35, 64 65, 65 66, 66 67, 67 68, 35 63, 35 36, 36 40, 36 37, 37 38, 38 39, 40 49, 40 41, 41 48, 41 42, 42 43, 43 44, 43 45, 45 46, 44 46, 46 47, 49 50, 50 51, 51 52, 52 53, 53 58, 53 54, 58 59, 59 60, 60 59, 59 61, 61 62, 54 55, 55 56, 56 57, 57 61, 1 2, 2 3, 3 4, 4 5, 5 6, 6 7, 7 8, 8 9, 9 27, 9 10, 27 30, 27 28, 28 29, 10 26, 10 11, 11 12, 12 13, 13 14, 14 15, 15 16, 16 19, 16 17, 17 18, 19 20, 20 21, 21 22, 22 23, 23 24, 24 25, 25 21, 18 10, 21 10, 62 50 5000
Affects	2 - Affects assign a1, a2; Select <a1, a2> such that Affects(a1, a2) 75 76, 69 71, 69 74, 5 7, 5 8, 5 30, 6 7, 6 8, 6 30, 7 8, 7 26, 14 22, 14 26, 14 12, 17 18, 17 12, 17 13, 17 22, 18 12, 23 23, 31 37, 31 52, 52 60, 55 51, 58 60, 58 51, 61 62, 62 51, 64 68, 65 66, 67 68 5000

We also tested semantically and syntactically invalid PQL queries to ensure that the SPA will handle these queries accurately, especially for differences in BOOLEAN return value for either type of invalid query.

Examples of PQL queries created:

Partition	Test case
Semantically invalid BOOLEAN return	1 - Invalid number of arguments for while pattern while w; Select BOOLEAN pattern w ("while" , _ , _) FALSE 5000
Semantically invalid tuple return	2 - Semantically invalid attribute for assign stmt assign a; while w; Select <a.varName, w> such that Next(a, w) 5000
Syntactically invalid BOOLEAN return	3 - Syntactically invalid explicit and assign a; if ifs1, ifs2; while w; Select BOOLEAN such that Parent(ifs1, a) and a ("x", _) 5000
Syntactically invalid tuple return	4 - Syntactically invalid tuple variable v; procedure p; Select <v>.varName such that Modifies(p, v) 5000

As mentioned in a previous section, invalid SIMPLE source programs were also created to test that recursive and cyclic procedures in the program will correctly cause the SPA to stop evaluating.

Lastly, we carried out stress testing. A different source code was created which also contains multiple procedures and heavily nested code. However, the PQL queries created for this source program contained 7-10 clauses, especially with multiple Affects clauses, to test if the SPA is able to complete such queries below 5 seconds. Furthermore, the queries also tested extreme cases where possible such as long tuples and names.

Examples of PQL queries created:

Partition	Test case
Multiple large table merging with long tuple	1 - Affects 1 assign a, a1, a2, a3, a4, a6, a7, a9, a10, a11, a12; Select <a, a1, a2, a7, a9> such that Affects*(a, a1) and Affects*(a1, a2) and Affects*(a3, a4) and Affects*(a4, a4) and Affects*(a6, a7) and Affects*(a7, a9) and Affects*(a9, a2) 22 22 22 22 22, 9 9 9 9 9, 47 47 47 47 47, 8 9 9 9 9, 8 9 9 8 8, 8 8 9 9 9, 9 9 9 8 9, 62 63 55 62 63, 8 9 9 8 9, 8 8 9 8 9, 13 22 22 22 22, 21 22 22 22 22, 17 22 22 22 22, 40 40 40 40 40, 61 62 68 62 63, 37 40 40 40 40, 62 63 68 62 63, 8 8 9 8 8, 36 40 40 40 40, 48 48 48 48 48, 8 8 8 8 8, 44 47 47 47 47, 61 63 55 62 63, 61 62 55 62 63, 9 9 9 8 8, 61 63 68 62 63 5000
Multiple such that clauses	2 - Variable name equals procedure name stmt s, s1; assign a, a1; while w; if ifs; variable v, v1; procedure p, q; constant c; read rd; print pn; call cl; Select <cl.procName, cl.stmt#, a.stmt#> such that Calls(_, p) and Next(_, _) and Uses(a, v) with v.varName = p.procName and v.varName = v1.varName and v.varName = cl.procName pattern a(_, _"(a+b)"_) pattern ifs(v, _, _) z 45 22, z 58 22, z 59 22, z 60 22, z 70 22 5000
Multiple pattern clauses	3 - Redundant pattern stmt s, s1; assign a, a1; while w; if ifs; variable v, v1; procedure p, q; constant c; read rd; print pn; call cl; Select <a, a1> pattern a(_, _) and a(_, _"a"_) and a(_, _"(a+(b))"_) pattern a(_, _"a+b"_) and a1(v, _"r"_) and a1(v, "r-1") and a1(_, _) such that Parent(_, a1) with c.value = 1 8 63, 22 63, 44 63 5000
Long tuple, with clause	4 - Long select tuple + symmetric with clause stmt s, s1; assign a, a1; while w; if ifs; variable v, v1; procedure p, q; constant c; read rd; print pn; call cl; Select <a, a, a, a, a, a, a, a, a, a, a, a, a, a, a> such that Follows*(_, _) and Calls*(_, _) such that Affects(_, _) and Parent*(_, _) and Modifies(a, v) and Modifies(a1, v1) with v.varName = "a" with "a" = v.varName with v1.varName = "b" 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8, 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13, 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 5000

The major contributing factors to the query time are analyzed as follows:

- We do note first the computation of Affects and Affects* cache tables are time consuming. However, we managed to achieve computation within 5 seconds and use the cached table for remaining Affects/Affects* clauses (test case 1 and 4 above) which saves a significant amount of time.
- A long tuple will cause result computation to be slow sometimes due to the need to do cross product among individual tables during the table merging process. We wrote test cases (e.g. case 1 and 4) to ensure our system can handle them within the time limit.
- A query with a large size of connected synonyms can sometimes run rather slowly (test case 2 and 3) as multiple table mergings are required.
- Large table size can also be a notable contributing factor as traversal though each of the large tables takes a noticeable amount of time. Nonetheless, our query optimization strategy has helped mitigate this issue so we are less concerned about this issue.

Part 2 – Project Management

3 Planning

Iteration 1		
Component	Team Member (Additional responsibilities, if any)	Allocation
Source processor	Lauren	● Implementation of tokenization for SIMPLE program ● Unit testing and system testing
Source processor	Yuxuan	● Validation of read/print/assign/if/while statements and procedure ● Population of read/print/assign/if/while statements and procedure ● Unit testing and integration testing
PKB	Zheng Wei	● Implement table template ● Populated nested Uses and Follows ● Build API for populator in Source Processor ● Unit tests and integration test
PKB	Zhenlin	● Implement table functions and entity sets ● Populated nested Modifies and Parent ● Implement pattern logic ● Work on code quality checking ● Unit tests for PKB
PQL	Xi Zhe	● Implementation of PQL query parsing ● Validation of PQL query ● Create Query object for query evaluator ● Unit testing and system testing
PQL	Guo Jun	● Implementation of query evaluator ● Handles the logic of finding all satisfied value

Iteration 2		
Component	Team Member (Additional responsibilities, if any)	Allocation
PQL	Lauren	● Validation of expressions in pattern clause ● Implement parsing of Select clause (Tuple, Boolean, attributes) ● Unit testing for query evaluator and system testing
Source processor	Yuxuan	● Implement parsing of multiple procedures for Source Porcesser ● Populate calls relationship ● Implement Control-Flow Graph
PKB	Zheng Wei	● Populate nested calls*, ModifiesP and UsesP, UsesS with calls and ModifiesS with calls ● Integration testing of DE with PKB ● Improve class diagram
PKB	Zhenlin	● Refactor code based on code review and create class diagrams ● Refactor all Pkb tables into index-based tables ● Implement tables and APIs for if/while/assign pattern clauses in PKB ● Implement entity-attribute related tables
PQL	Xi Zhe	● Implement parsing of multiple clauses ● Implement parsing of explicit and operation ● Implement parsing of with clauses ● Implement parsing of if/while pattern clauses ● Refactor Query class
PQL	Guo Jun	● Implement UsesP, ModfiesP, Calls and Calls* clause ● Implement with clause

Iteration 3		
Component	Team Member (Additional responsibilities, if any)	Allocation
PQL	Lauren	● Refactoring SP Tokenizer ● System tests
Source processor	Yuxuan	● Refactor populator and validator functions into classes with inheritance. ● Populate Next Relationship and perform integration testing for it.
PKB	Zheng Wei	● Refactoring ● System tests
PKB	Zhenlin	● Refactoring ● System tests ● Review code
PQL	Xi Zhe	● Refactoring ● Review code
PQL	Guo Jun	● Compute Next*/Affect/Affects* relationship ● Create cache class to store the complex relationship ● Grouping and sorting of clauses ● Refactor Query Evaluator ● Review code

Iteration 1			
Task id	Task	Activity id	Activity
1	Implement SIMPLE parser	1.1[P]	Research parsing strategies and AST (we can both write 1.1 in week 2/3)
		1.2[P]	Plan subcomponents for parser
		1.3[P]	Plan Front-end API for parser
		1.4[D]	Document front-end abstract API
		1.5[C]	Implement tokenization for read/print/assign/if/while statements
		1.6[C]	Implement AST
		1.7[C]	Implement validation and population for read/print/assign statements
		1.8[C]	Implement validation and population for if/while statements
		1.9[C]	Implement design extractor
		1.10[D]	Documentation for Iteration 1
		1.11[T]	Unit tests for tokenizer
		1.12[T]	Unit tests for AST
		1.13[T]	Unit tests for validation and population
		1.14[T]	Integration tests for populating PKB
		1.15[T]	System testing with autotester
2	Create PKB data structure	2.1[P]	Planning data structures for PKB, make design decisions for interactions with SP and PQL
		2.11[P]	Design APIs for SP to populate PKB
		2.12[P]	Design APIs for PQL to query PKB
		2.13[C]	Create template for tables
		2.14[C]	Create tables for entities and relations and their insertion/lookup methods
		2.15[C]	Populate nested Follows/ Parent relationship

		2.16[C]	Create APIs for PQL to query Follows / Parent
		2.17[C]	Create nested Uses / Modifies
		2.18[C]	Create postfix conversion for pattern queries; Implement pattern population / insertion / lookups
		2.19[C]	Populate nested Uses / Modifies relationship
		2.20[P]	Plan API for PQL queries, discuss with PQL side on API requirements.
		2.21[C]	Create APIs for PQL to query Uses / Modifies
		2.22[P]	Check for coding style issues
		2.23[C]	Move nested population into SP
		2.24[T]	Write test cases for entity tables
		2.25[T]	Write test cases for relation tables
		2.26[T]	Write test cases for public APIs exposed by PKB
		2.27[T]	Write integration test cases for design extractor with PKB
		2.28[T]	Write test cases for Pattern and entity sets
		2.29[T]	Write test cases for Populator and table functions
		2.30[D]	Documentation for Iteration 1
		2.31[D]	Documentation for API in google doc
3	Implement PQL Query Evaluator	3.1[P]	Plan subcomponents for PQL Query evaluator
		3.2[P]	Decide parsing strategy for PQL queries
		3.3[P]	Plan abstract API for PQL Query parser
		3.4[D]	Document abstract API for PQL Query parser
		3.5[C]	Implement Query class that would be used by query evaluator subcomponent
		3.6[C]	Implement Synonym class
		3.7[C]	Implement RelationshipToken class
		3.8[C]	Implement PatternToken class

		3.9[C]	Implement PQL query parsing for queries with synonym declarations and single Select clause
		3.10[C]	Implement PQL query parsing for queries with synonym declarations, single Select clause and single such that clause
		3.11[C]	Implement PQL query parsing for queries with synonym declarations, single Select clause, single such that clause and single pattern clause
		3.12[T]	Write unit test cases for PQL query parsing
		3.13[T]	System testing with autotester
		3.14[P]	Research on query evaluation strategy
		3.15[P]	Design API needed by Query Evaluator from PKB
		3.16[D]	Document Abstract API for PQL queries
		3.17[D]	Document Abstract API needed by Query Evaluator from Query Parser
		3.18[C]	Evaluation on queries with no such that/pattern clauses
		3.19[C]	Evaluation on queries with one such that/pattern clause
		3.20[C]	Implement InterTable class
		3.21[C]	Evaluation on queries with one such that clause and one pattern clause
		3.22[C]	Implement Clause class and all its children classes
		3.23[T]	Unit Testing for InterTable and Solver
		3.24[T]	Integration testing for Query Evaluator and Clause

Color code		Source processor	PKB	PQL
Legend:	[D] Documentation activity	[P] Planning/design activity	[C] Coding/ Implementation activity	[T] Testing activity

Iteration 2			
Task id	Task	Activity id	Activity
1	Extend SIMPLE parser	1.16[P]	Plan SP extensions for iteration 2
		1.17[C]	Implement parsing for multiple programs and call statements
		1.18[C]	Implement validation for cyclic/recursive calls
		1.19[T]	Test parsing and validation for multiple programs and call statements
		1.20[C]	Implement CFG
		1.21[C]	Refractor statement number from Tokenizer to Populator
		1.22[D]	Documentation for Iteration 2
		1.23[T]	System Testing
2	Extend PKB	2.1[P]	Plan PKB components for iteration 2
		2.2[C]	Create entity attribute tables and create APIs for SP/PQL to store/extract relevant attributes respectively
		2.3[C]	Refactor Pkb code from code review and feedbacks
		2.4[C]	Refactor Pkb tables to index-based tables
		2.5[T]	Write unit tests for refactored tables
		2.6[C]	Implement if/while pattern tables and APIs
		2.7[C]	Implement assign pattern tables and pattern helper classes
		2.8[T]	Write unit tests for pattern-related operations
		2.9[T]	Write unit tests for attribute table operations
		2.10[P]	Design class diagrams for Pkb
		2.11[C]	Implement inheritance structure for each table
		2.12[C]	Create new tables for calls and next and create API for PQL to query
		2.13[C]	Populate nested calls*
		2.14[C]	Create tables for ModifiesP and UsesP
		2.15[T]	Write integration tests for calls and calls* table

		2.16[C]	Populate nested ModifiesP and UsesP
		2.17[C]	Populate nested UsesS and ModifiesS with calls
		2.18[T]	Write integration tests for ModifiesP and UsesP
		2.19[T]	Write integration tests for UsesS and ModifiesS with calls
		2.20[D]	Documentation for Iteration 2
3	Extend PQL Query Evaluator	3.25[P]	Plan PQL components for iteration 2
		3.26[C]	Update Solver and InterTable to handle multiple return values
		3.27[C]	Parsing of multiple Such That and Pattern clauses
		3.28[C]	Parsing Explicit And operation
		3.29[C]	Parsing and validation of expressions in Pattern clause
		3.30[C]	Parsing Select clause with Tuple and Boolean value
		3.31[C]	Parsing attributes in Select clause
		3.32[C]	Parsing With clause
		3.33[C]	Parsing While and If Pattern clause
		3.34[C]	Refactor Query Class to use Clause
		3.35[C]	Query Evaluation for Calls
		3.36[C]	Query Evaluation for Boolean return value
		3.37[C]	Query Evaluation to return attribute values
		3.38[C]	Refactor query evaluation to use integers for representing design entities
		3.39[C]	Refactor Clause to query parser
		3.40[T]	Autotester for W8 Demo
		3.41[T]	Refactor query parsing unit tests with separate syntax and semantic error exceptions
		3.42[T]	Write unit tests for query parser
		3.43[T]	Write unit tests for query evaluator
		3.44[D]	Documentation for Iteration 2

Color code		Source processor	PKB	PQL
Legend:	[D] Documentation activity	[P] Planning/design activity	[C] Coding/ Implementation activity	[T] Testing activity

Iteration 3			
Task id	Task	Activity id	Activity
1	Refactor SIMPLE parser and testing	1.24	Refactoring SP Tokenizer
		1.25	System testing if/while pattern and Next/*
		1.26	System testing invalid queries
		1.27	Implement population for if/while pattern and Next
		1.28	Refactor SP Parser
		1.29	Integration testing for Next
		1.30	Unit testing for CFG
		1.31	Documentation for Iteration 3
2	Refactor PKB and testing	2.1	CFG related methods and tables
		2.2	Fix bugs in the code
		2.3	Refactor PKB based on code review
		2.4	Remove code duplications in the code
		2.5	Throw exceptions when appropriate
		2.6	System testing
		2.7	Stress Testing and Review Unit Testing
		2.8	Additional Unit Tests for tables and pattern helpers
		2.9	Documentation
3	Extend PQL Query	3.45	Review code for Query Parser
		3.46	Review code for Query Evaluator

	Evaluator and refactor	3.47	Refactor PQL Query Parser
		3.48	Refactor PQL Query Evaluator
		3.49	Implement cache class
		3.50	Group and sort the clauses
		3.51	Enhance unit testing for Query Parser
		3.52	Integration testing for new clauses
		3.53	Integration testing for cache class
		3.54	Integration testing for formatter class
		3.55	Documentation for Iteration 3

Color code		Source processor	PKB	PQL
Legend:	[D] Documentation activity	[P] Planning/design activity	[C] Coding/ Implementation activity	[T] Testing activity

3.1 Project plan

Iteration 1										
Team member	Activities									
	Week2		Week3		Week4		Week5		Week6	
Lauren	1.1	1.2	1.3	1.4	1.5	1.11	1.5	1.11	1.10	1.15
Yuxuan	1.1	1.2	1.6	1.12	1.7	1.13	1.8	1.14	1.10	1.15
Zheng Wei	2.1	2.12	2.13	2.13	2.13 2.14	2.15 2.16 2.17	2.20	2.21 2.23	2.24 2.25 2.26 2.27	2.30 2.31
Zhenlin	2.1	2.12	2.13	2.13	2.14	2.15 2.16 2.17 2.19	2.20 2.22	2.18 2.21	2.24 2.28 2.29	2.30 2.31
Guo Jun	3.1	3.14	3.16	3.17	3.18		3.19	3.20 3.21	3.22	3.23 3.24
Xi Zhe	3.1	3.2 3.3	3.5 3.6	3.9	3.7	3.10	3.8	3.11	3.12	3.13

Iteration 2						
Team member	Activities					
	Week7		Week8		Week9	
Lauren	3.25	3.29	3.31	3.42	1.23	3.44
Yuxuan	1.16	1.17 1.18	1.20		1.21	1.22
		1.19				1.23
Zheng Wei	2.1	2.12 2.13	2.15	2.14 2.16	2.17	2.18 2.19
						2.20

Zhenlin	2.1	2.2	2.3 2.4	2.6 2.7	2.10	2.20
		2.9	2.5	2.8	2.11	
Guo Jun	3.44	3.26 3.35	3.36	3.38	3.37	3.43
Xi Zhe	3.25	3.27 3.28	3.30	3.32	3.33 3.34	3.44

Iteration 3						
Team member	Activities					
	Week10		Week11		Week12	
Lauren	1.25		1.26		1.24	1.31
Yuxuan	1.27		1.28	1.29	1.28	1.31
			1.30			
Zheng Wei	2.1	2.1	2.5	2.9	2.9	2.6
				2.6		
Zhenlin	2.2	2.3	2.8	2.4	2.4	2.9
				2.2	2.7	2.7
Guo Jun	3.49		3.52 3.53	3.54	3.48 3.50	3.55
Xi Zhe	3.47		3.46	3.47	3.51	3.55

4 Test strategy

Our test strategy is to write unit test cases alongside the implementation of each subcomponent within the mini iterations, before carrying out integration testing between Source Processor and PKB and PKB and Query Processor. We used the testing framework Catch2 for unit and integration testing to automate the testing process and each subcomponent has their own test scripts. System testing was done via the autotester and the xml files generated after running each test suite were used to verify the results after each test.

Our team generally spent one week per unit/integration test, which included the time taken to implement the subcomponents. We typically spend one week performing system testing to ensure that all 3 components are working before the submission deadline but for Iteration 3, we spent about two weeks for system testing. Bugs were tracked in our telegram chat group and we would create pull requests after fixing the bugs and request other team members to review the changes made before merging it to the main branch. The average defect resolution time was around a few hours for minor issues and 2 days for major ones.

The following table represents a timeline of the test cases written and the person(s) in charge of writing them:

Iteration 1			
	Test case	Written on	Person(s) in charge
Unit Testing	SP: Tokenizer	Week 4	Lauren
	SP: Validator	Week 4	Yuxuan
	PKB: Tables / Sets	Week 4	Zhenlin
	PKB: Patterns	Week 5	Zhenlin
	PQL: Query Parser	Week 6	Xi Zhe
	PQL: Query Evaluator	Week 6	Guo Jun
Integration testing	SP & PKB: Populator	Week 5	Yuxuan
	SP & PKB: Design Extractor	Week 6	Zheng Wei
	PKB & PQL	Week 6	Guo Jun
System testing	Autotester	Week 6	Whole team

Iteration 2			
	Test case	Written on	Person(s) in charge
Unit Testing	SP: Validator	Week 7	Yuxuan
	PKB	Week 8	Zhenlin, Zheng Wei
	PQL: Query Parser	Week 7, 8, 9	Lauren, Xi Zhe
	PQL: Query Evaluator	Week 9	Guo Jun
Integration testing	SP & PKB: Populator	Week 7	Yuxuan
	SP & PKB: Design Extractor	Week 8, 9	Zheng Wei
	PKB & PQL	Week 9	Guo Jun
System testing	Autotester	Week 9	Lauren, Yuxuan

Iteration 3			
	Test case	Written on	Person(s) in charge
Unit Testing	SP: CFG	Week 11	Yuxuan, Guo Jun
	PKB	Week 12	Zhenlin
	PQL: Query Parser	Week 11	Xi Zhe
	PQL: Query Evaluator	Week 12	Guo Jun
Integration testing	SP & PKB: Next Rel	Week 11	Yuxuan
	PKB & PQL	Week 11	Guo Jun
System testing	Autotester	Week 10, 11, 12	Whole team

5 Coding standards

Before starting on this project, our team decided to follow the C++ coding standard provided by Google. As per the styling guide provided, as much as possible, we tried to ensure that the data members of a class end with a trailing underscore and that these variable names are named using underscore casing instead of other casing such as camel casing.

In terms of deviation, our team used “#pragma once” liberally as opposed to guards which was recommended by the style guide. This is because this was the default used in the project template and the team decided to stick with it instead of changing.

A short code example following the coding standards is given below.

```
class Token {
  public:
    TokenType type_ { TokenType::WHITESPACE }; // Initialised to WHITESPACE
    string text_ = "";

    bool operator==(const Token& t) const {
      return t.type_ == type_ && t.text_ == text_;
    }
};
```

6 Correspondence of the abstract API with the relevant C++ classes

During the implementation of SPA, our team generally used the same naming conventions in our abstract API and the C++ program. Since the naming of the API created during the planning phase was inline with the Google C++ style guide, we decided to use the same naming convention to avoid confusion between our team members. As a result of thorough planning, the abstract APIs that we had planned for translated almost perfectly into the relevant C++ classes and functions without much deviation from the original APIs. Below are 2 examples of the APIs having the same naming convention as the function in the C++ program.

LIST_INT GetModifiesStmtByVar(STRING a)
Returns all entities s such that Modifies(s, a) holds true. If no entity s such that Modifies(s, a) holds true, return an empty list.

```
vector<int> Pkb::GetModifiesStmtsByVar(const string& var) const {
  try {
    return modifies_variable_to_stmts_table_->GetValueByKey(var);
  } catch (exception& e) {
    return vector<int>{};
  }
}
```

SUCHTHATCLAUSE GetSuchThatClause()
Return the such that clause object. If there is no such that clause, return NULL.

```
std::vector<RelationshipToken> Query::GetSuchThatClause() {
  return Query::such_that_clauses;
}
```

Part 3 – Conclusion

7 Reflection

Iteration 1

Towards the deadline of iteration 1, the autotester failed to work properly for members of our team who were using the Mac operating system and this made it difficult for them to debug the integration testing. They encountered build errors on Mac while the system was able to build for Windows users. In future iterations, we hope to resolve these errors earlier.

Furthermore, since building SPA was new to us, we spent a lot of time working on the features and left integration and system testing towards the end. As we did not consciously write unit tests immediately after implementing features, many errors were also discovered in integration and system testing. This also affected our schedule as we had to rush to resolve these system testing errors.

After the submission for iteration 1, we encountered a problem with parsing the SIMPLE source program. We did not exit the program after encountering an invalid program, and the evaluation continued. We plan to write more test cases and do a more thorough job during unit testing in future iterations. We will also set aside more time for testing and debugging to ensure that bugs will be dealt with earlier before the submission deadline.

Moving on to the PKB subcomponent, the team should have checked the priority of the requirements before designing the components. For iteration 1, we should have prioritised efficient query retrieval from the PKB. However, we ended up using other data structures that were not as efficient. We later revamped the data structures used and this led to wasted time and effort. This redundant work can be avoided in future iterations by carefully evaluating the requirements before commencing on the design implementations.

For the PQL query evaluator component, we realised that we should have utilised more software engineering principles during the implementation. Due to the time constraint, we focused more on the code functionality and as a result, neglected some of the engineering principles which resulted in a messier codebase. For instance, we should have abstracted more of the lengthy functions into smaller functions. Furthermore, we should have also used OOP for the pattern clause and “such that” clause which would help with extendibility and reduce code duplication.

Although the team encountered multiple difficulties along the way, there were still positives that we could take away. Our usage of GitHub issues as well as the Kanban board for product management helped to ensure that our tasks were completed on time and that everyone was aware of the progress of others. While the workload for iteration one was relatively heavy, we were also able to divide the workload amongst the three subteams equally. More importantly, everyone was hardworking and passionate and contributed to the development of our SPA. As a result, we were able to successfully finish the tasks that were required for iteration 1 as well as for the demo in week 5 (with a little help from our TA).

Iteration 2

After experiencing build failures and autotester issues for members of our team who were using the Mac operating system in iteration 1, from iteration 2 onwards, members who were using Mac made sure to test the build as well as the autotester whenever a new commit was merged into the master branch. This allowed us to rectify any errors and issues immediately on Mac whenever they occurred. It also allowed us to narrow down the potential areas that might have caused the build failures and autotester failures.

In addition, in iteration 1, we left the testing late till the end and it resulted in a lot of errors being uncovered at the last minute. In iteration 2, our group made an effort to update unit and integration tests each week as we built the features. We also allocated sufficient time for system testing so that bugs could be discovered and fixed in a timely manner.

Finally, we were also able to get feedback from the Oral Presentation and our TA to refactor our code. For Source Processor, we moved the counting of statement numbers from Tokenizer to Populator so as to adhere to better software engineering principles. For the PKB component, the structure of inheritance for tables was updated to allow for better extensibility while increasing separation of concerns. For PQL Parser, the token class was removed and clause class is used to avoid additional conversion from token to clause.

Overall, this second iteration was executed well and we managed to finish most of the features that we had set out to accomplish for this iteration. Using the lessons learned from the mistakes made in iteration 1, we put measures in place to make sure that the same mistakes were not committed again and this has proved beneficial for iteration 2. For iteration 3, we will again be conducting a post-mortem to make sure that issues and mistakes made for iteration 2 will not be repeated.

Iteration 3

For this final iteration, we mainly focused on refactoring, system testing and optimisation. Most of the features required for the complete SPA were done in iteration 2 and this gave us more time to focus on the other priorities. Since correctness occupied a huge percentage for the final iteration, we decided to spend more time on system testing as compared to iteration 2 and we started system testing earlier and allocated more members to work on this. We had different members working on the correctness of the new relationships introduced in iteration 2 and members working on the validity of the queries as well as members who were working on queries with multiple clauses. All these helped to stress test our system as well as to ensure robustness and correctness. Thanks to our TA, we also managed to obtain test cases created by her group in the past semester and we made use of these additional test cases to ensure that our SPA was working as expected.

Code quality was yet another component that held a significant weightage. As such, while working on the system testing, we also made sure to review and refactor the code such that we were following good software engineering practices and principles. During the process,

we realised that it might be difficult to spot the errors for the components that we have written. As such, in addition to the code review conducted by our TA, we also held our own mini internal code review where we tried to spot potential coding violations for the other components as given by the rubrics.

Moving on to optimisation, due to the time constraints as well as the fact that we were already storing multiple tables in the PKB, we decided to focus mainly on efficiency for the computation of queries. We made use of certain principles such as memoization such that most API calls to the PKB from the PQL could be done in O(1) time. On the PQL side, we also made use of sorting to group the clauses to help improve the efficiency of queries.

To close this, all of these were only made possible by the members. Everyone played their part and did what was required of them on time and this helped to significantly reduce the blockage as no one member was waiting for the work of another member for weeks. This was also clearly noticeable during the system testing of SPA. Whenever bugs were found, members who were familiar with the relevant components would immediately fix the bugs and this allowed the system testing to proceed rapidly since the issue would be solved within a day to two days.

8 Appendix

8.1 SP Abstract API

Abstract APIs provided by Source Processor:

BOOL Validate()
Check procedure calls for cyclical/recursive calls and calling of non-existent procedures. Return: ● TRUE: if the source code is semantically correct. ● FALSE: otherwise

VOID Populate(PKB pkb)
Extract necessary program design entities and basic relationships and populate them into the PKB.

VOID PopulateNestedRelationships()
Extract nested relationships and populate them into the PKB

8.2 PKB Abstract API

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, INT key, LIST_INT value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, INT key, LIST_STRING value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, INT key, INT value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, INT key, STRING value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, STRING key, LIST_STRING value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddInfoToTable(TABLER_IDENTIFIER table_identifier, STRING key, PAIR_INT value)
Add key value relationship to table based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddEntityToSet(ENTITY_IDENTIFIER entity_identifier, INT entity_val)
Add the entity to a set based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddEntityToSet(ENTITY_IDENTIFIER entity_identifier, STRING entity_val)
Add the entity to a set based on the identifier Return: ● True if added successfully ● Throws an exception otherwise

BOOL AddCfg(CFG cfg)
Adds a CFG to the list of CFGs stored in the PKB Return: ● True if added successfully

BOOL IsRelationshipHolds(RELATIONSHIP_TYPES rel_type, INT stmt_1, INT stmt_2)
Depending on the rel_type passed in, it can check and return whether stmt_1 is a parent of stmt_2 or if stmt_1 calls stmt_2 and so on Return: ● True if yes ● False otherwise

BOOL IsRelationshipExists(RELATIONSHIP_TYPES rel_type)
Depending on the rel_type passed in, it can check and return whether the relations table contains any key value pairs or not Return: ● True if yes ● False otherwise

LIST_INT GetRelFirstArgument(RELATIONSHIP_TYPES rel_type, INT second_arg_idx)
Depending on the rel_type passed in, it either returns the parent of second_arg_idx if any or the procedure that called second_arg_idx and so on Return: ● True if yes ● Empty list otherwise

LIST_INT GetRelSecondArgument(RELATIONSHIP_TYPES rel_type, INT first_arg_idx)
Depending on the rel_type passed in, it either returns the children of first_arg_idx if any or the procedure that is called by first_arg_idx and so on Return: ● True if yes ● Empty list otherwise

LIST_PAIR_INT GetRelArgumentPairs(RELATIONSHIP_TYPES rel_type)
Depending on the rel_type passed in, return a list of pairs of all the keys and values for a particular relationship type Return: ● List of pair of integers ● Empty list otherwise

SET_INT GetAllStmtsWithPattern(STRING pattern, BOOL is_exact)
Get all statements with the relevant matching patterns. Depending on whether is_exact is true or not, the patterns can be matching exactly or not Return: ● Set of integers

SET_STRING GetAllPatternVariablesInStmt(INT stmt_no, TABLE_IDENTIFIER table_identifier)
Get all pattern variables for the relevant statement number Return: ● Set of strings

SET_INT GetAllEntity(ENTITY_IDENTIFIER entity_identifier)
Get all the items for an entity type Return: ● Returns a set of all item of the specified entity type based on entity_identifier ● Otherwise return an empty set

MAP GetNextInternalMap()
Gets the internal map for the next table Return: ● Returns the internal map for the next table in stored in the PKB

LIST_PAIR_INT_STRING GetAllIndexStringPairs(INDEX_TABLE_TYPE idx_table_type)
Get all the pairs of index and its corresponding variable name for the specified index table Return: ● Returns a list of all pairs of indexes and variable names with one-to-one mapping ● Otherwise return an empty list

LIST_PAIR_STRING_INT GetAllStringIndexPairs(INDEX_TABLE_TYPE idx_table_type)
Get all the pairs of variable name and its corresponding index for the specified index table Return: ● Returns a list of all pairs of indexes and variable names with one-to-one mapping ● Otherwise return an empty list

STRING GetStringByIndex(INDEX_TABLE_TYPE idx_table_type, INT idx)
Get the string name corresponding to an index for the specified index table Return: ● Returns the string mapped from the index given ● Otherwise return an empty string

INT GetIndexByString(INDEX_TABLE_TYPE idx_table_type, STRING entity_name)
Get the integer index corresponding to the name for the specified index table Return: ● Returns the index mapped from the name string given ● Otherwise return invalid index

TABLE GetTable(TABLE_TYPE type)
Get the relevant table Return: • Returns the table specified

8.3 Query Parser Abstract API

LIST_SYNONYM GetAllUsedSynonyms()
Returns the list of synonyms that have been used in the query.

LIST_ATTRREF GetAttrRef()
Returns the list of attribute references for the Select result clause. It can return an empty list for the case where the query selects BOOLEAN.

LST_PTR_CLAUSE GetClauses()
Returns the list of pointers to the Clause objects. It can return an empty list if there are no clauses in the query.

BOOLEAN GetBoolean()
Returns true if the query is selecting BOOLEAN, false otherwise.

BOOLEAN IsSemanticallyValid()
Returns true if the query is semantically valid, false otherwise.

8.4 Query Evaluator Abstract API

LIST_STRING EvaluateQuery(Query query, Pkb b)
Takes in query and pkb as the arguments and returns a list of strings that satisfy all clauses of the given query. Can return an empty string if there is no value that can satisfy the query.

Solver

LIST_STRING Solve()
Find all values that satisfy all predicates by applying a series of merging and filtering.

8.5 Extension to SPA (Iteration 2)

We propose an extension of a negation to 'such that' and 'with' clauses in PQL. In the following sections, we will elaborate on the specification, implementation details, possible challenges and benefits to SPA.

Definition

The current SPA supports 'such that' and 'with' clauses by only returning entities which match with every clause in the query. This limits the possibilities for the user during querying, but with negation, there will be greater flexibility and the user will be able to get entities which do not match with certain clauses in the query.

The PQL grammar would have to be extended for both of these clauses. For a 'such that' clause, to extend it, we add an optional '!' sign to the start of every relRef.

```
relCond : relRef ( 'and' relRef )*

relRef:  [ '!' ] Follows | FollowsT | Parent | ParentT | UsesS |
UsesP | ModifiesS | ModifiesP | Calls | CallsT | Next | NextT |
Affects | AffectsT
```

For a 'with' clause, the existing grammar is:

```
with-cl : 'with' attrCond

attrCond : attrCompare ( 'and' attrCompare )*

attrCompare : ref '=' ref
```

To extend it, we add an optional '!' sign to the operation in attrCompare.

```
attrCompare : ref '=' ref | ref '!=' ref
```

Changes required to existing system

Changes are required only for PQL. Source processor will not be affected as there are no changes to the SIMPLE source language and PKB will still store the normal relationships. Additional components will not be required and the existing subcomponents will be sufficient to support the change.

For Query Parser, the new relationship must be parsed as there is a change to the PQL grammar. It will need to parse '!' after 'such that' (or 'and'), whereas it will expect '=' or '!=' for 'with' clause. It will then create clauses with `is_negated` set to true.

The existing 'such that' clause and 'with' clause can be extended to support the new changes by adding an attribute `is_negated` to indicate whether the clause is negated. For Query Evaluator, only the 'such that' clause and 'with' clause need to be modified.

Furthermore, unit testing for PQL, integration testing of PQL and PKB, and system testing must also be updated to test the new feature added and ensure that it works as expected. A similar approach already used for system testing (as elaborated in previous sections) can be extended for the new queries.

Implementation details

For the Query parser to parse the new such that clause, it just checks whether '!' appears after 'such that' (or 'and') keyword. It can then continue to parse the rest of the part without any modifications. On the other hand, the Query Parser will need to expect '=' or '!=' for 'with' clauses. This can be simply done by adding one extra check.

For Query Evaluator, to deal with negated clauses, it first needs to get all possible values and subtract those that hold the relationship.

For example, if the clause contains only one synonym like `!Follows(s, 5)`, it will get all possible values of s first, then remove values that are in `GetStmtBefore(5)` to get the complement domain.

If the clause contains two synonyms e.g. `!Parent*(s, s1)`, it will apply a Cartesian product between the domains of `s`, `s1`. Then, it will remove the values that are in `GetAllParentPairs()`.

Unit testing and integration testing for PQL should be updated consistently during the implementation of this change to ensure that it works as expected, for instance testing the changes made with only one 'such that' / 'with' clause first and later testing multiple negated clauses. System testing with complex programs will be implemented after the unit/integration tests to ensure that the extension made work as expected with the existing SPA implementation.

Possible challenges and mitigation plans

One possible challenge is the speed of query evaluation since for negated clauses we need to do extra work to get the values that satisfy the clause. The bottleneck would then be the speed to subtract two sets and apply a Cartesian product.

An optimisation that we can consider is to use multi-threading to handle the sets operation but it would require more work and careful implementation.

Moreover, the Query Evaluator can first go through all clauses to check if any contradiction exists. If so, it could return an empty/false result without having to evaluate the clause. However, finding contradictions is not trivial and more research is needed.

Lastly, the Query Evaluator can cache the result of the negated clause so that multiple occurrences of the same negated clause would only require one slow calculation.

Benefits to SPA

The benefit of having a negation relationship is that the user will have a greater flexibility for writing queries on PQL and will be able to determine more information through queries that were not previously possible using the Advanced SPA requirements.

For instance, the negation of the 'such that' clause can be used to find variable names in the program that are not used by a given procedure, or a list of constants used that exclude a given value. Below is an example with the negated 'such that' clause:

```
variable v; procedure p;

Select v such that !Uses(p, v)
```

Another example is that the user will be able to use the negation of 'with' clause to find procedures which do not use a certain given variable. This would not be possible with the current SPA requirements as users would only be able to query procedures that are using a given variable. Below is an example with the negated 'with' clause:

```
variable v; procedure p, p1;

Select p such that Uses(p1, v) with p.procName != p1.procName and
v.varName = "x"
```