NUS National University of Singapore

School of Computing

CS3203 - Software Engineering Project

Static Program Analyzer

Iteration 1 Project Report

Team 05

Table of Contents

1. Scope of the Prototype Implementation

In this iteration of the project, we have developed a functional prototype of a Static Program Analyzer (SPA). Our prototype allows users to enter SIMPLE source program which is taken in as input. SPA then parses the source program, extracting design abstractions and stores it in the Program Knowledge Base (PKB).

Afterwards, SPA allows users to ask questions about the program, accepting queries written in Program Query Language (PQL). The PQL queries are then processed and relevant information of design abstractions is extracted from the PKB and returned to the user.

Our SPA prototype fulfills the Basic SPA requirements for Iteration 1:

- Parser able to read/recognize full specification of SIMPLE language
- PKB encapsulates information about basic relationships in SIMPLE
- Query Preprocessor able to parse basic PQL queries and identify invalid PQL queries
- Query Evaluator able to evaluate valid PQL queries and returns correct results

2. Development Plan

After much discussion and careful planning of the project scope, we divided our team of 6 into 3 sub-teams, where each pair is responsible for implementing one component of the SPA. This allocation allows us to have a partner to discuss inter-component issues together, since two brains work better than one. As for testing and documentation, the team decided to work on them together as a whole. The table below shows the components each member is responsible for, as well as their designated roles.

Parser	PKB	PQL
Zeke (Testing IC)	Ee Ter (GitHub IC)	Grant (Team Leader #1)
Cheng Yi (Team Leader #2)	Jeun Hoe (Documentation IC)	Christabel (Mini-iterations IC)

In week three, we learnt about the software development cycle. Our team decided to adopt a breadth-first iterative SDLC so that we could start performing integration testing early, without having to complete all functions of the SPA. We broke down the time we had for Iteration 1 into 4 mini-iterations, each lasting a week. We have listed the planned goals and designated activities for each mini iteration in the tables provided below.

Mini Iteration #	Goals for Iteration
Mini Iteration 1 (Week 3)	Setting up SPA components with functionality for Follows and Parent. Write unit test.
Mini Iteration 2 (Week 4)	SPA should be able to answer Uses, Modifies, and Pattern queries.
Mini Iteration 3 (Week 5)	Perform integration testing Write system acceptance test cases
Mini Iteration 4 (Week 6)	Perform system testing, final adjustment to source code. Documentation of APIs and Report Writing.

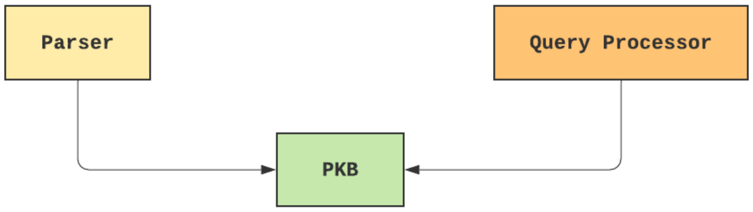
Activity Table

Component	Person	Activity
Parser	Cheng Yi	• Create Parser class to identify statement type and passes relevant data to ParserController. • Create ParserController class to handle each statement by passing each statement to Validator and Logic. • Create Logic class to extract relevant information from source code and calls respective API from PKB to store the information.
	Zeke	• Create Validator class to check for all syntactic expressions. This includes var_name, proc_name, cond_expr etc. • Write unit tests for container statements (if and while) and assignment statements and combinations of the above
PKB	Ee Ter	• Maintain GitHub repo • Create RelationshipTable class to store design abstractions • Implement API functions used by Parser to store uses and modifies relationships • Implement API functions used by Query Processor to retrieve selected uses and modifies relationships • Write unit tests for parent, uses and modifies API calls
	Jeun Hoe	• Set up data structures to store variables and procedure names • Implement API functions used by Parser to store and get variables, procedures, follows, parent and pattern relationships • Implement API functions used by Query Processor to • Implement function to generate postfix expressions for pattern queries • Write DRAFT report for Iteration 1
Query Processor	Grant	• Implementation of QueryEvaluator consisting of: o Evaluation methods depending on format of clause o Methods to call PKB APIs o Methods to filter between results from the evaluation of each clause o Methods to format the input for each clause • Write unit tests for static methods in Query Processor component • Write integration tests between Query component and PKB component • Write system tests involving no, single and multiple clauses
	Christabel	• Mainly in charge of the QueryPreprocesser: parsing of PQL Queries into a PQLQuery object for the QueryEvaluator. • PQLQuery object contains and allows information transfer such as the attributes declared in the query, and the Clauses in it. • Implement a Clause struct and a child Pattern struct to facilitate reference to the clauses in a PQLQuery. • Query Validation: create methods to catch invalid PQL queries before passing to the evaluator

3. SPA design

3.1. Overview of Architecture diagram

Our SPA is made up of 3 separate components: Parser, PKB and Query Processor. We have attached a UML class diagram below to show the overall architecture of our SPA.



The Parser component is responsible for taking in SIMPLE source programs, and checks the program for any errors (syntax, grammar) before extracting relevant design abstractions. It then stores design abstractions and relationships in the PKB using API calls provided by the PKB.

The PKB serves as a database for the input SIMPLE source program. It stores basic information of the source program (variables, procedures) as well as design abstractions and basic relationships (follows, parent, uses, modifies & pattern). It provides API calls to both the Parser and the Query Processor to populate the database and to retrieve information for query results.

The Query Processor parses user queries written in PQL, and then extracts relevant information from the PKB using API calls. It then processes the information from PKB and evaluates the query, returning the result to the user.

3.2. Design of SPA components

3.2.1. SPA Front-End Design

Parser

The parser will read the entire source SIMPLE program line by line and convert it into a string.

Example:

```
1 procedure ABC {
2     read x;
3 }
```

will be converted into “procedure ABC {read x;}” and passed into ParseString() along with the pkb.

After which, ParseString will be called on this string and will look for the first position of the delimiters (‘{‘, ‘;’ and ‘}’) while maintaining information to track the statements. These positions are stored respectively as pos_left, pos_semicolon and pos_right respectively.

The minimum of the three positions will be identified, and everything before that position will be processed first. In the above example, ‘{‘ will be the first to be detected. Hence, “procedure ABC” will be processed first.

Once the statement type has been identified, the statement will have its identifier and any whitespace (including tabs and spaces) removed and passed to the ParserController along with other information that needs to be stored in the PKB.

Example:

ParseString() will then identify this statement as a procedure statement by matching the first 9 characters, and identify "ABC" as the procedure name by checking everything from the 10th character to the position of '{'. ParserController's HandleProcedure is then called, which passes the procedure_name and pkb into the method.

Any messages thrown by the ParserController will be caught and ParseString() will return -1, which will be caught by Parse() and the program will be exited.

ParserController

The ParserController will first validate the syntax of the given statement by calling methods from the Validator class.

In our example, we will call the IsNAME method on the procedure name that is passed in.

If the method returns true, it means that there is no problems with the procedure name and logic's method will be called. Otherwise, there is a problem with the procedure name, and a message is thrown.

Validator

For Procedure/Print/Read/Call statements, the implementation was done by using regex checking to see if the given input complies with SIMPLE NAME syntax.

For Assignment statements, we used a left to right approach for validating the statement. Using regex, we identified the first operator (-+*/%(), if the encountered operator was a '(', we would use a stack to identify the contents of the '()' and assign it to the left-hand side, else anything on the left of the found operator was assigned to the left-hand side. We validated the attained left-hand-side against SIMPLE syntax and used recursion for the remainder of the statement on the right-hand side of the operator.

The hardest of them all was the conditional statements, since there are many ways of nesting conditions. The idea that was eventually settled upon was to iterate through the passed statement and maintain a stack to track the '(', ')'. Once the stack was empty, we would have our left- and right-hand side of the expressions. Should the string remained unchanged, it would suggest that the conditional expression did not have '&&' or '||'. This process was repeated recursively until the simplest form of conditional expression was left. For example, the conditional expression (((x *4) != 2) && (y <= z)) will first have the outer encasing bracket removed to become ((x * 4) != 2) && (y <= z). The Validator will then identify the conditional expressions inside like such: LHS: ((x * 4) != 2) and RHS: (y <= z). Then it will repeat the process until it evaluates the expression (x * 4). The final brackets will be removed and the LHS and RHS of the operator * is evaluated. x being evaluated as a NAME and 4 being evaluated as a const.

Next was validating the attained conditional expression against the SIMPLE syntax for relation expressions using regex matching to once again split the relation expressions into a left- and right-hand side. We then validated each side of the relation expression against the SIMPLE syntax for expression.

If Validation is successful and the ParserController is finished with its task, the Parser will update any data structure it uses to track information on the statements. Should Validation fail, exceptions are thrown in the statement being validated and the parser will stop parsing as per the requirements.

Logic

If Validation is successful, the ParserController will ask the Logic component to perform design abstraction on the statement to retrieve the information to be passed to the PKB. The Logic component oversees calling the PKB APIs to store information into PKB.

The logic handles each statement by its statement type and extracts the design abstractions. Notable implementation was for assignment statements and conditional statements. Again, using regex to accomplish by iterating through the string looking for any operators as a delimiter.

After the variables are extracted, along with the additional information passed in with controller; current parent, statement number and statement type, the Logic component will set the relationships that the statement number could possibly produce and populate the tables inside the PKB. For example, a read statement will call the following PKB APIs: SetParent, SetStatement and SetModifies.

3.2.2. PKB design

Data Structures

When designing the PKB, our team had to decide an appropriate way to store relationships of the SIMPLE program. Instead of using an AST to store these relationships, our team opted for using a solution using tables and variable matching. The pros and cons of each decision are mentioned below

AST:

- **Advantages**
  - Stores all relationships in the program very clearly
  - Only required 1 AST for the entire program for Basic SPA
- **Disadvantages**
  - Extra time spent in creating and implementing the AST
  - Longer traversal time of AST just to calculate simple queries
  - Likely to use more memory in system

Table and Variable Matching (Hash Tables):

- **Advantages**
  - Easier to implement
  - Existing data structures available in C++
  - Very efficient as accessing from hash table
  - Separate table for separate relationships
  - Basic & Advanced SPA can be implemented without an AST at all
- **Disadvantages**
  - Additional work required to implement pattern matching

The PKB was designed with close referencing to the PQL design entities to make implementation of API function calls easier. The table below shows an overview of data structures in the PKB for each specific function.

Relationship stored	Data Structure
Variables and procedures	Set<VARIABLE> variableSet Set<PROCEDURE> procedureSet
Statement Types	HashTable<STMT_TYPE, STMT_NUM> statementTypeTable
Parent/Parent*	HashTable<STMT_NUM, STMT_NUM> parentTable HashTable<STMT_NUM, List<STMT_NUM>> childTable
Follow/Follow*	HashTable<STMT_NUM, STMT_NUM> followsTable HashTable<STMT_NUM, STMT_NUM> followedByTable
Uses	HashTable<STMT_NUM, Set<VARIABLE>> usesTable HashTable<VARIABLE, Set<STMT_NUM>> usedByTable
Modifies	HashTable<STMT_NUM, Set<VARIABLE>> modifiesTable HashTable<VARIABLE, Set<STMT_NUM>> modifiedByTable
Pattern	HashTable<STMT_NUM, String> assignmentToPatternTable HashTable<String, List<STMT_NUM>> patternToAssignmentTable

<u>Storage of Statements, Variables and Procedures</u>

As PQL queries support the declaration of synonyms, we created two sets to maintain all variables and procedures within a SIMPLE program. We chose to use a set as we can easily check for duplicate procedures or variables, which are not allowed in SIMPLE programs.

As the PQL also allows for declaration of statement types (assign, while, if etc.), we created a statementTypeTable, to store all statement numbers that correspond to a statement type.

<u>Parent/Parent*</u>

In the parent table, the key value represents a statement number, while the value represents the parent's number. In the child table, the key value represents a statement, while the value is a list containing all children of the statement.

The parent table allows us to check the direct parent (i.e. Parent (x, 7)) of any statement with 1 access. If we wish to check the parent* relation upwards (i.e. Parent* (x, 7)), we can recursively trace the parent table to check parents of parents.

The child table is useful if we wish to check a child or all children of a statement using the same method of tracing explained above.

<u>Follow/Follows*</u>

To capture the follow/follows* relationship, we created two tables: followsTable & followedByTable. If the relationship Follows(x, y) is true, x is stored as a key, and y is a value in the followsTable. The reverse relationship is stored in the followedByTable. As a variable could be set to either the first or second argument (i.e. Follows(2, x) or Follows(x, 2) in this relationship, we store the inverse relation to allow for efficient querying. The same is done for most other relationships.

This child table mentioned above is also used for checking the follows* relationship. As the List<STMT_NUM> contains all statements in the same nesting level, we traverse this list to check all statements that Follows*. We can get this list by first checking the parent of the statement, and then obtaining the list by checking children of parent.

Special Case: Checking follows across if/else blocks

As the above implementation checks for follows by looking at all children of its parent, it could potentially identify that statements in an else block Follows a statement in an if block. As such, we have created a elseBlockSet to contain all statements in else blocks. For follows queries, we check if the statement is in/not in an else block and filter out the statements using this set.

Uses/Modifies

Since Uses and Modifies are treated identically by the PKB with two similar sets of data structures and methods, this paragraph will only discuss Uses. Uses data is stored in a class containing two HashMaps, one to store the forward relationship and one for the inverse relationship for fast lookup. The forward map has the program variables stored as values and the inverse one has the variables stored as the key. The PKB is called with the statement number that uses a variable, and the class stores it in both the forward and inverse table. The PKB then recursively checks the parent of the statement and stores this info for the parent as well. For retrieval, the class just looks it up in either the forward or the inverse table depending on the query.

Since a statement or procedure can use multiple variables, and a variable can be used by multiple statements or procedures, the value of the abovementioned HashMap must be a data structure holding multiple items. If a List is used to hold all the values, there would be duplicates as a statement or procedure can use a variable multiple times and vice versa. Then the code must check for duplicates before insertion. Thus, a HashSet was used instead to hold the values. The resulting signature of the forward and inverse relationship tables is HashMap<Key, HashSet<Value>>.

Pattern/Pattern*

As mentioned earlier, we chose not to use an AST to do matching of patterns as we felt that querying an AST would take too much time. However, our team still had to find a workaround an AST to handle pattern queries as they use tree matching. Our team chose to use the Shunting Yard Algorithm to generate postfix expressions which we stored in 2 tables: assignmentToPatternTable and patternToAssignmentTable. To check if a given expression matches the AST, we simply convert the input string into a postfix expression and then perform token matching using the tables above.

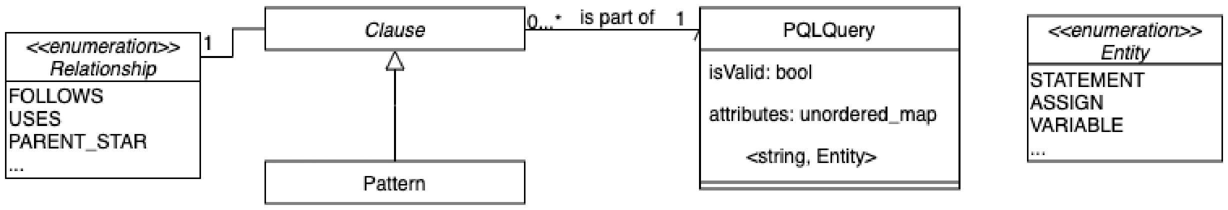
API Calls

API calls were designed for the Parser to populate these data structures in the PKB as well as for the Query Processor to retrieve relevant information from the PKB. They are documented below in section 7.

3.2.3. Query Processor

Query Preprocessor

The Query Processor comprises of the Query Preprocessor and the Query Evaluator. The Preprocessor is responsible for the parsing and the construction of PQLQuery objects, which will then be passed into the Evaluator for query evaluation. This includes the design of Clause and its storage in the PQLQuery object, which are supposed to ease the information transfer from the query to the evaluator. During the Furthermore, query validation is also done at this sub-component, so the Evaluator will only handle valid queries, making the whole Query Processor component more efficient in evaluating the results from the queries given by the user. The figure below gives an overview of the QueryPreprocessor sub-component.



Clause Design

Clauses (such that) include information about its parameters, stored as left and right respectively, as well the Relationship (i.e. Follows, Uses), which is an enumeration. Pattern is made to be a child of Clause and stores information about the assignment attribute for pattern checking, as well as the variable and expression. These objects are constructed as c++ struct objects instead of classes, so that the Evaluator can easily access the parameters of the clause or pattern without having to call the getter all the time, since struct variables are public by default. There are methods to check if the parameters are underscore by using IsLeftUnderscore() and IsRightUnderscore() which returns a boolean. There is an extra variable for the Pattern struct, IsExpressionPartial, which is true if the expression given is a partial expression, else it will be false. Evaluator will use this variable to call the corresponding API call from the PKB.

PQLQuery Design

A PQLQuery object is designed as a class and consists of an attributes list, attribute to be selected, and the clauses present. In our project scope, attributes refer to the temporary variables that the user has defined to be used in the queries. For example, “assign a1; variable v;”, a1 and v are both attributes with the Entity ASSIGN and VARIABLE respectively. All attributes declared are tied to an Entity enumeration type and is stored in unordered_map<string attributeName, Entity> to ease reference to the Entity type in the Evaluator. Furthermore, the PKB stores the type of the statement using Entity, which will more efficiently allow the Query Evaluator to evaluate queries.

A vector of clauses is stored in the PQLQuery object as vector<unique_ptr<Clause>>, which is a vector of unique_ptr which points to a Clause or the Pattern child object. This implementation will prevent object slicing of the child Pattern clause when storing it in the vector. When the Query Evaluator later references a Pattern object, we will use dynamic_cast to retrieve the child Pattern object. This method preserves all the information stored in the child Pattern class while allowing easy iteration of the vector for evaluation of clauses. This design can also be extended to allow sorting based on the type of clause in future iterations.

Information in the PQLQuery object are made to be easily accessible by the Query Evaluator with methods and checks. Some important methods are:

- GetAttributeToSelect(): string, which returns the attribute name declared for selection.
- GetSelectEntity(): Entity, which tells us what is the Entity of the select attribute, so the evaluator can filter accordingly.
- IsLeftAttribute(Clause) and IsRightAttribute(Clause), which returns a boolean whether the parameters in the clause are attributes declared in the PQLQuery. This will be checked against the unordered_map attributes stored in the PQLQuery object.
- GetLeftAttributeEntity(Clause) and GetRightAttributeEntity(Clause), which returns the Entity that is related to the parameter’s declared attribute in the PQLQuery. This will be checked against the unordered_map attributes stored in the PQLQuery object.

Query Parsing

Apart from the design of the components, the main task of the Preprocessor is to parse the given query and to construct the corresponding PQLQuery object for the evaluation of the query in the Query Evaluator. This will be done with the ParseQuery(string query) method. For any given string query, the parsing logic will be broken down in a few parts, namely: attribute declaration, query selector, looping such that or pattern clause, which will be executed in the mentioned order. An example is:

ParseQuery(“assign aa; stmt s1, s2; Select s1 pattern aa ("y", _"x +1"_) such that Parent*(s1, s2)”)

1. Attribute declaration
    - An unordered_map<string, Entity> attributeMap is initialised to facilitate storing of attributes and its corresponding Entity.
    - The parser looks out for a semicolon ‘;’ which signals the end of an Entity’s declaration of its corresponding attributes. The first word will be converted into an Entity enum using the StringToEntity(string) method.
    - Next, the program will look out for any commas ',' present in the attribute substring. If a comma exists, it will loop every attribute declared for that Entity, and store it in the attributeMap. Else, store the only attribute and its Entity. Loop steps b. and c. until no more attributes are present.
2. Query Selector
    - The parser looks out for the next word that comes after the “Select” keyword, that will be defined as the selectAttribute.
    - Now, a PQLQuery object is initialised, taking in attributeMap in the previous step and the selectAttribute string.
3. Looping through the clauses
    - The program now checks whether the next keyword is a such that clause or a pattern clause with its corresponding keywords, “such that” or “pattern” respectively. The parser will then proceed to the corresponding loop for the clauses. This section will be in a while loop until there is no more clauses to parse.

    a. Such that clause
        - The parser makes use of the brackets and commas to separate important information in the clause. The substring that comes before the open bracket ‘(‘ and after the “such that” keyword is stored as the relationship in string. Clause constructor will convert it to the corresponding Relationship enum.
        - The string between the open bracket and comma is the leftString and the one between the comma and close bracket ‘)’ is the rightString.
        - Now that we have the information needed for Clause creation we will initialise the Clause object with the leftString, rightString, and Relationship in string. Next, we use the PQLQuery::SetClause(<unique_ptr<Clause>>) method to store the clause in the vector of clauses in the PQLQuery object.

    b. Pattern clause
        - After the “pattern” keyword is detected, we look for the next word in the clause before the open bracket, that will be the assignAttribute.
        - Similarly, for the pattern clause, the string between the open bracket and comma is the variable and the one between the comma and close bracket is the expression.
        - With the information gathered from the parsing, we now create the Pattern clause with the variable, expression, and assignAttribute. Relationship is automatically set to PATTERN upon construction of the Pattern object. Next, we use the

PQLQuery::SetClause(<unique_ptr<Pattern>>) method to store the clause in the vector of clauses in the PQLQuery object.

4. Once the query is empty, parsing is done and the method will return the PQLQuery object that was created, with all the relevant information.

An important part of the query parser is to correctly identify crucial keywords regardless of spaces. In order to make sure the parser does this correctly, we implemented the TrimLeft(string), TrimRIght(string) and Trim(string) methods, without using the external c++'s boost library. These trim methods help to remove extra and unnecessary whitespaces in the program, especially when tokenising keywords, attributes and parameters. This therefore ensured the correctness of information parsed from queries.

<u>Query Validation</u>

In addition, the Query Preprocessor is responsible for the catching of invalid queries. To facilitate query validation during the parsing of queries, we created an InvalidQueryException which is to be thrown whenever an invalid query is detected. A PQL query can become invalid from various reasons. Some examples are invalid syntax, incorrect case for certain keywords (e.g. "follows" instead of "Follows"), incorrect Entity passed into the parameters of the clauses, and usage of undeclared attributes. Query validation follows closely with the execution of query parsing, as we want to detect any invalid query as soon as possible. Query validation in the Preprocessor is done on two different levels. The first type of validation is the checking of the PQL Query's syntax. The second is checking the clause's validity.

To aid with the query validation process the following methods are written and is used in both levels of query validation:

- IsAttributeDeclared(attrMap, attrName), which returns true when the attribute with attrName is declared in the attrMap, and false otherwise.
- IsNumber(string), which returns true if it contains number only, and false otherwise.
- IsVariable(string), which returns true if it is a valid var_name from the SIMPLE language syntax, and false otherwise.

For the first level of validation, which is the checking of PQL Query syntax, we go by the following rules. If any of the rules are not fulfilled, InvalidQueryException is thrown:

- No extra symbols or sections of the query that do not belong in the PQL.
- All spelling of keywords, such as in attribute declaration, select, or relationships should be correct and in the right case.
- Select attribute should have already been declared in the attribute map.
- All clauses should only take in two parameters, delimited by a comma.

The next level of validation is to check the clause validity. Checking for invalid clauses in the Preprocessor will save the Evaluator a lot of time and effort. The Evaluator will only need to deal with valid queries, and this will make the Query Processor component much more efficient and easier to implement. To do that, the following methods are written with their corresponding validation rules:

- CheckPatternValidity(attrMap, pattern) which checks for validity in a pattern clause
    - Checks whether the pattern's assign attribute is declared in the attribute map, and if the corresponding entity is ASSIGN.
    - Variable stored in the pattern should be true for IsVariable(), or an attribute stored in the map with VARIABLE Entity.

    - Check for the validity of the pattern's expression, rejecting all empty expressions and mismatched quotes or underscores.
- CheckClauseValidity(attrMap, clause) which checks for validity in a such that clause. It checks and makes sure that the parameters passed in are valid for the respective relationship.
    - For all relationship types, if an attribute is used as parameter, make sure the attribute is declared and the corresponding Entity type is correct for that clause relationship. For example, Follows, Follows*, Parent, and Parent* should take in statements only; Uses and Modifies' right parameter can only be a variable.
    - Uses and Modifies' left attribute cannot take in an underscore.
    - Statement numbers used in the parameters must have IsVariable() == true.
    - Variables used in the parameters must have IsVariable() == true.

At the end of the parsing, if an InvalidQueryException is caught, the PQLQuery will be set as an invalid query. This is done by changing the isValid variable in the PQLQuery object to false. The Query Evaluator will then use a method, IsQueryValid() to check for the validity of the query.

<u>Query Evaluator</u>

Given a valid query in the format "Select [synonym] [clause 1] [clause 2] …", Query Evaluator breaks down each clause to be evaluated individually and independently of other clauses. Take note that clauses for Iteration 1 can be broken down into the following classification:

1. Such that clause
    a. Direct relationship
        i. e.g. Follows (7, s), Parent (7, _), Uses (7, v), Modifies ("Example", _)
        ii. The goal is to evaluate the possible values of attribute s or _ (unnamed attribute)
    b. Inverse relationship
        i. e.g. Follows (s, 7), Parent (_, 7), Uses (s, "x"), Modifies (p, "x")
        ii. The goal is to evaluate the possible values of attribute s or _ (unnamed attribute)
    c. Bidirectional relationship
        i. e.g. Follows (s1, s2), Parent (s1, s2), Uses (s, v), Modifies (p, v)
        ii. The goal is to evaluate possible values for both unknown attributes
    d. Simple relationship
        i. e.g. Follows (1, 2), Parent (1, 2), Uses (7, "x"), Modifies ("Example", "x")
        ii. The goal is to evaluate whether the clause is true or false
2. Pattern clause
    a. Left attribute is fixed
        i. Right expression is an exact match, e.g. pattern a ("x", "y")
        ii. Right expression is a partial match, e.g. pattern a ("x", _"y"_)
        iii. The goal is to evaluate possible values of attribute a
    b. Left attribute is unknown
        i. Right expression is an exact match, e.g. pattern a (v, "y")
        ii. Right expression is a partial match, e.g. pattern a (v, _"y"_)
        iii. The goal is to evaluate possible values of both unknown attributes a and v

There can also be several cases of combinations of clauses for iteration 1, and they are as follows:

1. No clause
    a. e.g. stmt s; Select s
    b. The expected result is to return all possible values of s
2. 1 such that clause
    a. With no unknown synonyms
        i. e.g. assign a; Select a such that Uses (“Example”, “x”)
        ii. The expected result is to return:
            I) All possible values of a, if the clause is evaluated to be true
            II) An empty list, if the clause is evaluated to be false
    b. With 1 unknown synonym
        i. e.g. stmt s; Select s such that Follows (3, s)
        ii. The expected result is to return all possible values of s that fulfil the clause (i.e. values of s that results in the clause evaluating to be true)
3. 1 pattern clause
    a. With 1 unknown synonyms
        i. e.g. assign a; Select a pattern a (“x”, _ “y”_)
        ii. The expected result is to return all possible values of a that fulfil the pattern clause (i.e. a should be an assignment statement that modifies “x” and has a partial match for the expression on the right)
    b. With 2 unknown synonyms
        i. e.g. assign a; variable v; Select a pattern a (v, _)
        ii. The expected result is to return all possible values of a and v that fulfil the pattern clause (i.e. a should be an assignment statement that modifies v)
4. 1 such that clause and 1 pattern clause
    a. With no common synonyms
        i. e.g. variable v, v1; assign a, a1; Select a such that Uses (a, v) pattern a1 (v1, _“x”_)
        ii. The expected result is to return all possible values of a, given that they fulfil the Uses clause and there exists values of a1 and v1 that allows the pattern clause to evaluate to be true
    b. With 1 common synonym that is the select entity
        i. e.g. variable v, v1; assign a; Select a such that Uses (a, v) pattern a (v1, _“z”_)
        ii. The expected result is to return all possible values of a, given that they fulfil the Uses clause as well as the pattern clause
    c. With 1 common synonym that is not the select entity
        i. e.g. variable v; assign a, a1; Select a such that Uses (a, v) pattern a1 (v, _“x”_)
        ii. The expected result is to return all possible values of a, given that they fulfil the Uses clause, given that the values of v that they are “bound” to also fulfil the pattern clause
    d. With 2 common synonyms
        i. e.g. variable v; assign a; Select a such that Uses (a, v) pattern a (v, _“z”_)
        ii. The expected result is to return all possible values of a, given that they fulfil the Uses clause, given that both those values of a and the values of v that they are “bound” to also fulfil the pattern clause

The implementation for query evaluation works by evaluating each clause independently of other clauses. The result is then “updated” after the evaluation of each clause. Upon the completion of evaluating all clauses, the final result is returned. The logical flow of the implementation is explained step-by-step below. For purposes of the explanation, we shall use the following example (from 4c above):

variable v; assign a, a1; Select a such that Uses (a, v) pattern a1 (v, _"x"_)

1. Upon initialisation, the Query Evaluator creates a new query results hashset (format: set<map<string, string>>) for keeping track of valid results as each clause is being evaluated.
2. When the Query Evaluator receives a PQLQuery object from the Query Preprocessor, it takes note of the following:
   a. The attribute to select: a
   b. The vector of clauses to be evaluated: Uses (a, v) and pattern a1 (v, _"x"_)
3. Based on the attribute to select, a list of all possible values for that attribute is retrieved by calling the relevant PKB API method.
   a. Query Evaluator checks a against the attribute list in the PQLQuery object and understands that a is an assignment statement.
   b. Query Evaluator has a GetAllOfEntity(entity) method, which either calls the GetAllVariables() PKB API, or (as in this query case) the GetStatementType("assign") PKB API to retrieve the list of assignment statements from PKB.
4. Each value in this list is inserted into a map<string, string>, where the key is "a" and value is the value from the list. This map is put into the query results hashset.
   a. Each map represents a single valid result (i.e. a single possible combination of attribute values) up to the point of evaluation.
   b. Let's say that the PKB API call returns statements 1, 3, 5 and 7 as assignment statements in the SIMPLE source program. Then, the query results hashset = [{"a": "1"}, {"a": "3"}, {"a": "5"}, {"a": "7"}].
5. Next, we iterate through the vector of clauses to evaluate the result for each clause independently. The result for each clause should be stored in the format of a clause results hashset (i.e. set<map<string, string>>), similar to the query results hashset.
6. For each clause, identify if it is a pattern clause or such-that clause. If it is a such-that clause, move on to step 7. If it is a pattern clause, move to step 8.
7. To evaluate a such that clause, first identify the direction of the relationship, as defined above. Then call the corresponding PKB API(s) to retrieve the relevant lists. As described in step 5, a final step is to store the results in a clause results hashset that "binds" the values between attributes.
   a. For direct relationships and inverse relationships, call the corresponding PKB API according to the table below.
      i. After retrieving the list, it is essential to filter against a list of statements of the attribute type as we are only concerned with possible values of that attribute.
      ii. The list of statements of the attribute type can be retrieved using the GetAllOfEntity(entity), which calls either the GetStatementType(type) or GetAllVariables() PKB API, depending on the entity type.
      iii. For example, if the clause is "such that Uses (a, "x"), we will call the GetUsedByStatements("x") PKB API, as well as the GetStatementType("assign") PKB API. We will then need to filter the former list against the latter list, to ensure that we end up with only assignment statements that use "x".

Relationship	**PKB API to call for direct relationship**	**PKB API to call for inverse relationship**
Follows	GetFollows(left_stmt)	GetFollowedBy(right_stmt)
Follows*	GetAllFollows(left_stmt)	GetAllFollowedBy(right_stmt)
Parent	GetChildren(left_stmt)	GetParent(right_stmt)
Parent*	GetChildrenOfChildren(left_stmt)	GetAllParents(right_stmt)
Uses	GetUsesVariables(left_entity)	GetUsedByStatements(right_entity)

Modifies	GetModifiesVariables(left_entity)	GetModifiedByStatements(right_entity)

b. For bidirectional relationships, there are two unknown attributes to find possible values for. Hence, we will approach the evaluation by first getting a list of all possible values for the entity of the left-hand-side attribute, by calling the GetStatementType(type) PKB API. For each of these values, we then treat it as a direct relationship and evaluate using the steps above.
   i. In our general example query above, in order to evaluate "such that Uses (a, v)", the Query Evaluator will first call the GetStatementType("assign") PKB API to retrieve a list of all assignment statements ["1", "3", "5", "7"].
   ii. Then, for each assignment statement, we will call the GetUsesVariables(x) where x is the statement number in string format.

c. For simple relationships, the evaluation logic is trivial. Since we are looking to evaluate whether the relationship is true or false, we call the PKB API for direct relationship, using the left-hand-side value as the argument. Upon receiving the list of possible values for the right-hand-side entity, we check if the right-hand-side value can be found in the list. If so, the relationship is true, else it is not.

d. Finally, an important step is to store the results in a clause results hashset in the same format as the query results hashset, i.e. set<map<string, string>>. The reason for using a map as a data structure is so that the values of one attribute are "bound" to the corresponding values of another attribute. Given so, each map then represents a single possible combination of values of unknown attributes in the given clause.
   i. For direct and inverse relationships, the clause results hashset will only store values for the single unknown attribute.
   ii. For bidirectional relationships, the clause results hashset will store maps for every entry, where each entry is a value for an unknown attribute that corresponds to the other entries in the map. For example, after evaluating "such that Uses (a, v)", the clause results hashset = [{"a": "1", "v": "x"}, {"a": "1", "v": "y"}, {"a": "5", "v": "x"}, {"a": "7", "v": "z"}].
   iii. For simple relationships, if the relationship is true, the clause results hashset will be a copy of the query results. If the relationship is not true, the clause results hashset will be an empty set.

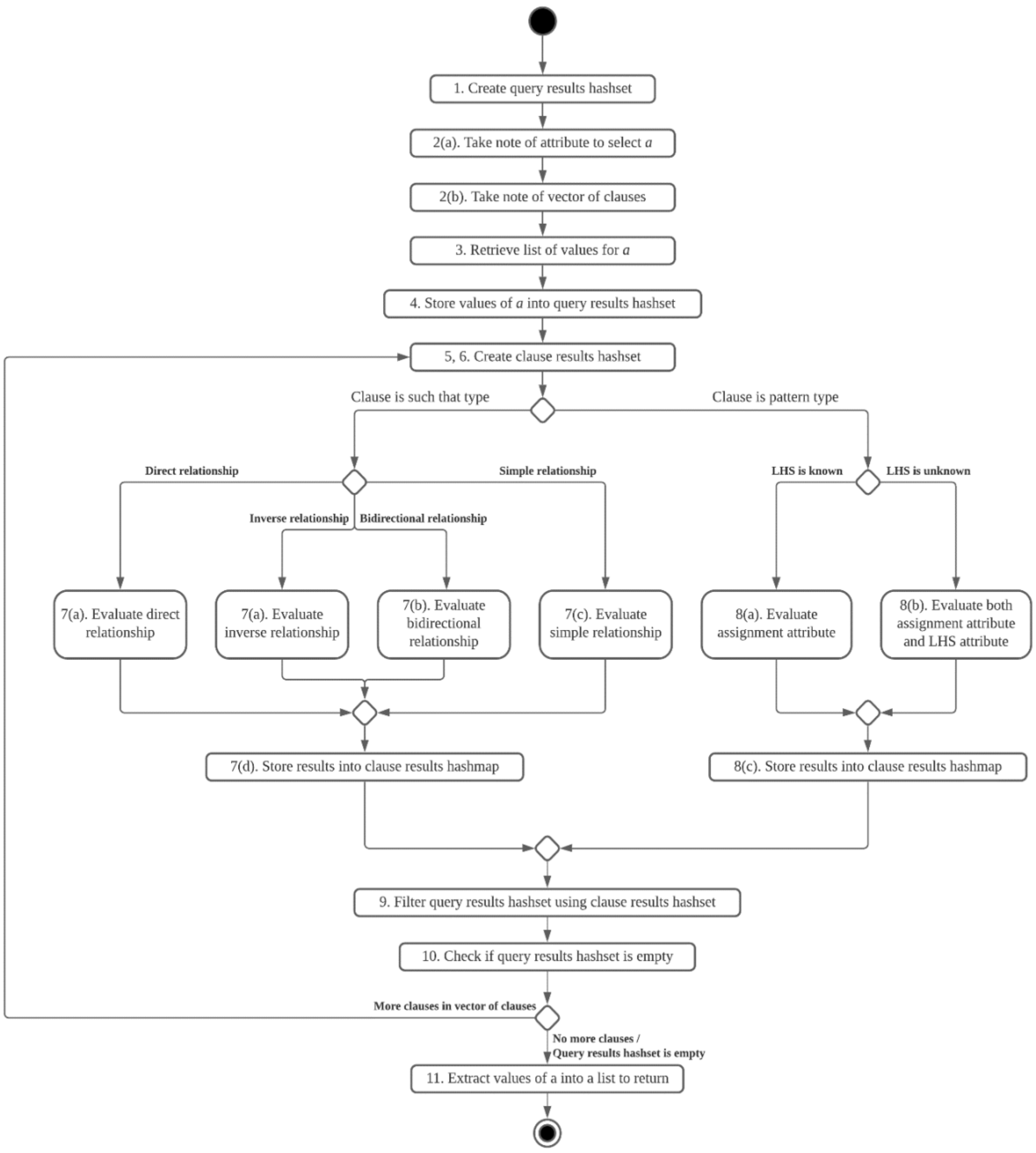
8. To evaluate a pattern clause, we need to consider two cases: when the left-hand-side argument (i.e. the modified variable) is known or unknown. Then we proceed to call the corresponding PKB API in order to retrieve relevant lists. As described in step 5, a final step is to store the results in a clause results hashset that "binds" the values between attributes.
   a. For cases where the left-hand-side argument (i.e. the modified variable) is known, we only need to retrieve values of one attribute (a). We will call the GetPatternStatements(left, right, IsExpressionPartial) PKB API in order to get the list of assignment statements that fulfil the pattern clause. We will then put each of these statements into a map with key "a" and statement number as the value, and put these maps into a clause results hashset.
   b. For cases where the left-hand-side argument (i.e. the modified variable) is unknown, we need to retrieve values of two attributes – the assignment statements and their corresponding modified variables. For example, in our sample query above, the pattern clause is "pattern a1 (v, _"x"_)". In order to evaluate this pattern clause, we first call the GetPatternVariables("x", IsExpressionPartial=true) PKB API, returning a list of all values for v, i.e. variables that are modified in assignment statements. Then, for each of these values for v, we call the GetPatternStatements(value_of_v, "x", IsExpressionPartial=true) PKB API to retrieve the corresponding values of a1, i.e. assignment statements that correspond to the value_of_v.

   c. Similarly, we will store the results in a clause results hashset in the same format as the query results hashset, i.e. set<map<string, string>>. We put each pair of a1 and v into a map (representing a possible combination of values), and collate these maps into a clause results hashset. Then, the clause results hashset = [{“a1”: “3”, “v”: “x”}, {“a1”: “1”, “v”: “y”}]

9. The final step in the evaluation of each clause is to filter (and update) the “global” query results hashset using the clause results hashset that we have generated in either step 7 or 8. After this step, the “global” query results hashset should contain a number (possibly 0) of maps, each representing a valid result up to the current clause that has been evaluated.
   a. This is done via the FilterResults(clause_results_hashset) method. This method compares between the maps in both the query results hashset as well as the clause results hashset and attempts to merge them using the MergeCombinations(map_from_query_results_hashset, map_from_clause_results_hashset) method.
   b. The FilterResults and MergeCombinations methods provide the logic that removes entries (maps) from the query results hashset do not have a corresponding result in the clause results hashset, as such an occurrence indicates that the clause cannot be fulfilled for that given set of attribute values.
   c. For entries from the query results hashset that still fulfil the incoming clause, the map is updated with the incoming values (if any) of the newly associated attribute. For example, for our given query example above:
      i. The query results hashset at step 4b = [{“a”: “1”}, {“a”: “3”}, {“a”: “5”}, {“a”: “7”}].
      ii. After evaluating the such that clause at step 7d(ii), the clause results hashset = [{“a”: “1”, “v”: “x”}, {“a”: “1”, “v”: “y”}, {“a”: “5”, “v”: “x”}, {“a”: “7”, “v”: “z”}].
      iii. After filtering the results, the query results hashset = [{“a”: “1”, “v”: “x”}, {“a”: “1”, “v”: “y”}, {“a”: “5”, “v”: “x”}, {“a”: “7”, “v”: “z”}]. Take note that the entry {“a”: “3”} has been removed from the query results hashset, as there are no values of v that correspond the value “3” of a.
      iv. After evaluating the pattern clause at step 8b, the clause results hashset = [{“a1”: “3”, “v”: “x”}, {“a1”: “1”, “v”: “y”}].
      v. After filtering the results, the query results hashset = [{“a”: “1”, “v”: “x”, “a1”: “3”}, {“a”: “1”, “v”: “y”, “a1”: “1”}, {“a”: “5”, “v”: “x”, “a1”: “3”}]. Take note that the entry {“a”: “7”, “v”: “z”} has been removed, as there are no possible combination of values in the clause results hashset that correspond to the value “z” of v. In addition, all other still-valid clauses are updated with the associated values of a1.

10. Repeat steps 6 to 9 for every clause in the vector of clauses. At every point after evaluating each clause, check if the query results hashset is empty. If so, terminate the clause evaluation process early, as it suggests that there is no possible combination of attribute values that fulfil the clauses that have been evaluated so far. Hence, as a logical deduction, there is no possible combination of attribute values that can fulfil all clauses.
11. After all clause evaluations are complete, from the query results hashset, extract only the values for the attribute that we are concerned with (the attribute to select) and put them into a list. As a final step, the Query Evaluator will then remove all duplicates in the list and return this list.

The following activity diagram shown below provides a summary of the logic above:



1. Create query results hashset
2(a). Take note of attribute to select *a*
2(b). Take note of vector of clauses
3. Retrieve list of values for *a*
4. Store values of *a* into query results hashset
5, 6. Create clause results hashset
Clause is such that type
Clause is pattern type
Direct relationship
Simple relationship
Inverse relationship
Bidirectional relationship
LHS is known
LHS is unknown
7(a). Evaluate direct relationship
7(a). Evaluate inverse relationship
7(b). Evaluate bidirectional relationship
7(c). Evaluate simple relationship
8(a). Evaluate assignment attribute
8(b). Evaluate both assignment attribute and LHS attribute
7(d). Store results into clause results hashmap
8(c). Store results into clause results hashmap
9. Filter query results hashset using clause results hashset
10. Check if query results hashset is empty
More clauses in vector of clauses
No more clauses / Query results hashset is empty
11. Extract values of a into a list to return

The benefits of this implementation of the Query Evaluator, as compared to the previous implementation (discussed in Section 4.3) are:

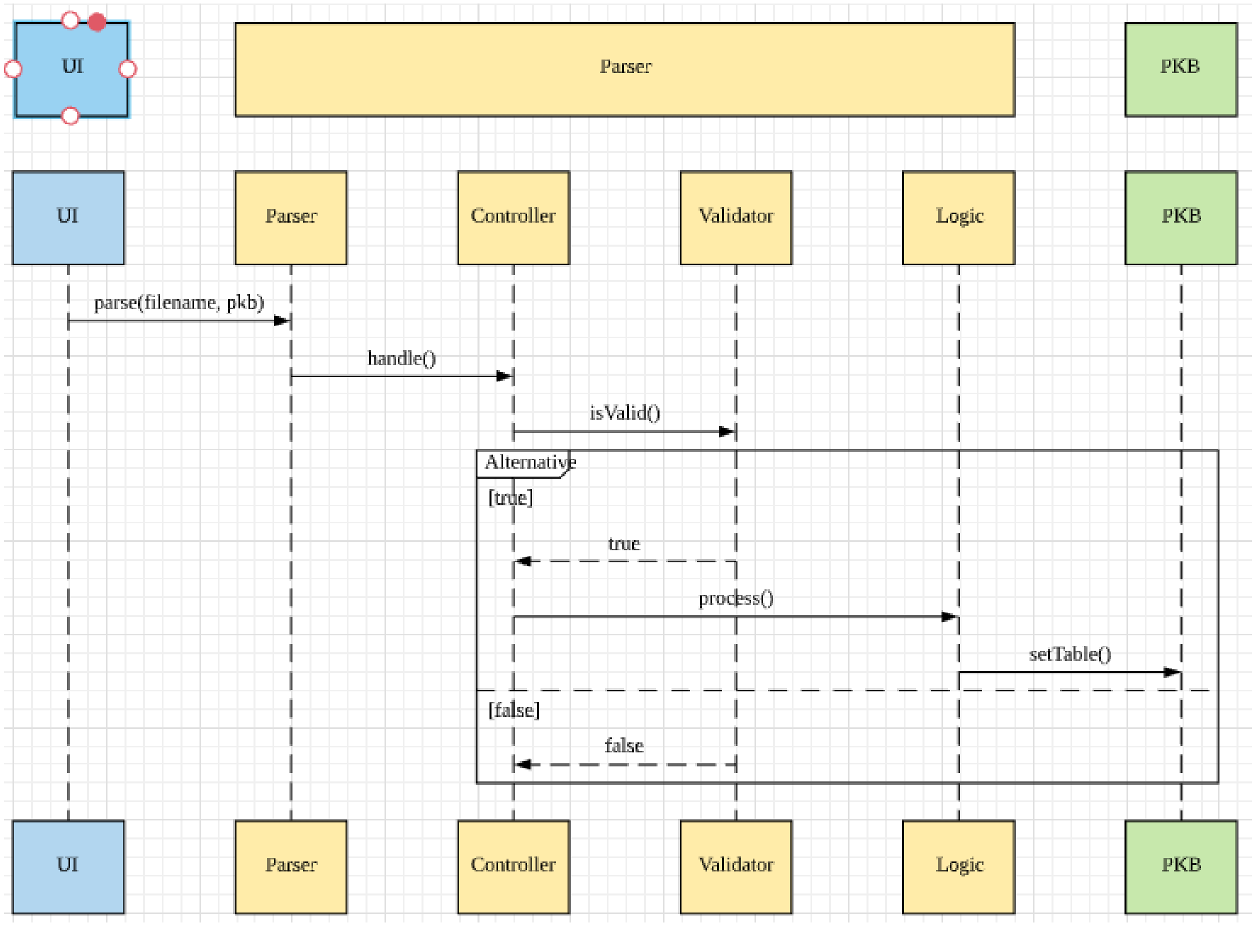
- The "binding" between attribute values are accurately recorded and tracked, since they are stored into the same map. If a clause invalidates a value for one of the attributes, all combinations that contain that value are no longer correct. In terms of implementation, this translates to entire maps (that contain the value) being removed from the query results hashset.

- Using a set to represent the solution space, instead of a list or vector, ensures that there are no duplicate results. This can reduce the time required to evaluate the query marginally.

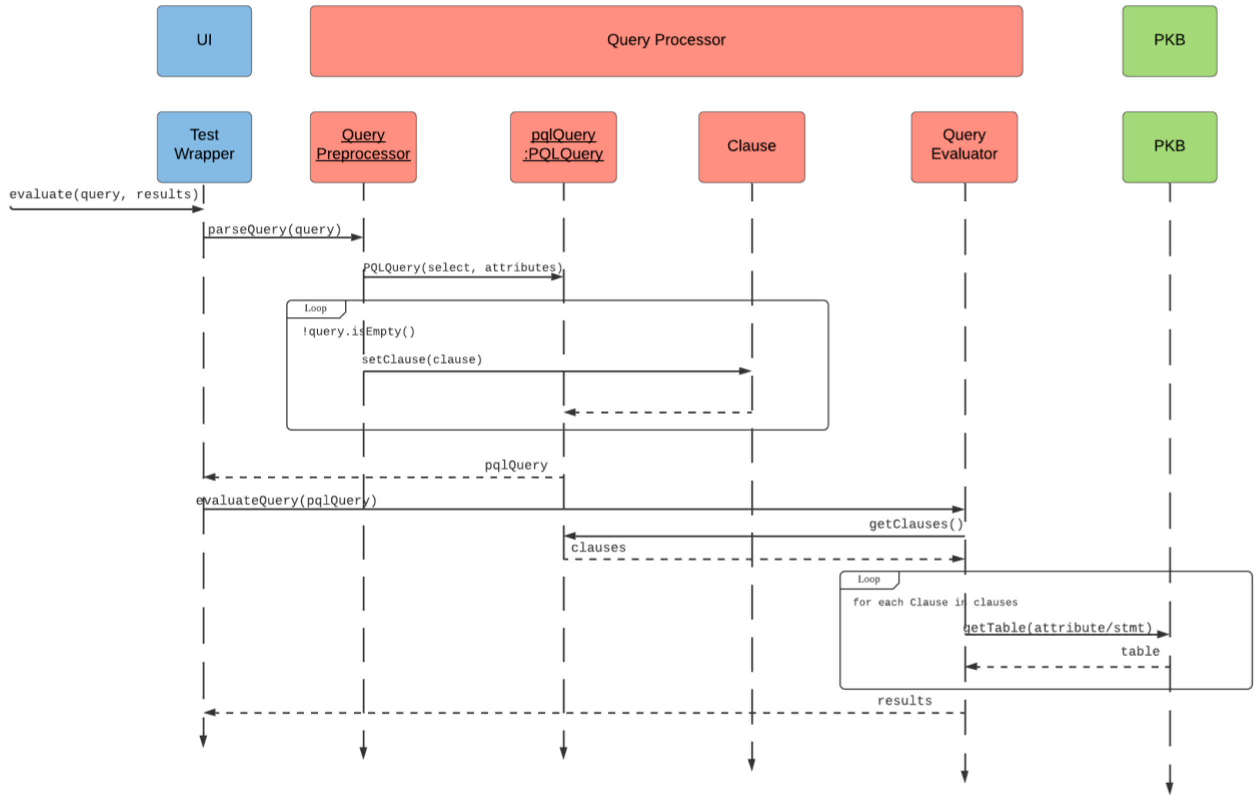
3.3. Component interactions

The PKB is the central knowledge base for the Parser to input data, and for the Query to retrieve information. The two figures below show the component interactions with the PKB, using a sequence diagram to show the flow of class level interactions, starting from the user input. It also shows how the sub-components interact with each other within the component.

Parser and PKB



PQL and PKB



The team designed both sequence diagrams above during the project planning phase. This is so that all team members would have a shared understanding of the logical flow of the program. Throughout the implementation phases, we referred to the respective diagrams to ensure that our components were adhering to the planned design, and data was being passed correctly from one component to another. The sequence diagrams proved to be effective not just as a means of explaining our program design, but also as a basis for effective communication and discussion as well. After every given mini-iteration, we ensured that our components adhered to the sequence diagrams and hence were well-aligned with the planned design. The result of this was consistent integration between all components (and sub-components) throughout the implementation phases.

4. Documentation and coding standards

4.1. Google C++ style guide

Our team has referenced from the Google C++ style guide as our coding standard with certain modifications, such as using camel case for naming of variables and methods. Modifications we have made to the style guide are mentioned below:

- **Class definition:** Our class definitions starts with the private section (which is default in c++) followed by protected:, then public: section. Omit sections that would be empty.
- **Function Length:** As it was impossible for certain methods and algorithms in our implementation to be shorter than 40 lines, we did not follow this restriction.
- **Filenames:** Filenames are in camel case, with each first letter of the word capitalised, or the capitalised first letter of every word. E.g. ProgramKnowledgeBase.cpp or PKB.cpp
- **Variable Names:** Variable names follow the camel case with the first letter being lowercase. E.g. myFirstVariable.
- **Function names:** Function names are in Pascal case, starting with a capital letter and have a capital letter for each new word, with no underscore, e.g. MyExcitingFunction(). Regular functions have mixed case; accessors and mutators may be named like variables.
- **ENUM names:** Enumerators are named using full caps, separated by an underscore for more than 1 word. E.g. ENUM_NAME.
- **Line length:** at most 80 chars
- **Indentation:** indent 4 spaces at a time.
- **Conditionals:** The if and else keywords belong on separate lines. There should be a space between the if and the open parenthesis, and between the close parenthesis and the curly brace (if any), but no space between the parentheses and the condition.

4.2. API naming

During the planning phase of the project, our team has carefully considered the different cases of SIMPLE program and PQL queries. Using what was taught in our lectures, we spent time to discuss what tables will be needed between the different components of the SPA project.

We created tables for the Parser->PKB and PQL->PKB abstract APIs on a team word document. This includes consideration of different cases. Following the exact naming and parameters, this allows us to continue our implementation on our respective branches and integrate properly afterwards. API functions were named such that it is intuitive for the reader to understand what the function does from its name alone e.g. GetParent(child), GetAllVariables().

4.3. Insufficient previous implementation of Query Evaluator

The importance of accurately representing the query details becomes apparent when we consider that the representation of these details should cover all aspects and relationships between attributes and clauses. This write-up has been included so that the learning points can be shared with other developers.

The following algorithm design for the Query Evaluator was implemented in a previous mini-iteration. It has since been updated to the implementation in Section 3.2. This change was made due to the realization that this previous implementation (below) was not sufficient to cover the case for queries with 2 clauses with 1 common synonym that is not the select entity. This is because it does not record (and track) the binding between attribute values, which is essential for query evaluation.

In order to record this binding characteristic between attributes, the implementation was updated to represent the solution as a tree of values instead. For purposes of explanation, we shall use the following query example:

while w; Select w such that Follows (w, 7)

The previous (insufficient) implementation of query evaluation follows as such:

1. Save all synonyms into an attributes hashmap:
    a. attributes map: { "w" : WHILE }
2. Evaluate "Select w", refer to attributes map from step 1, recognise that it's asking for while, call PKB API for a list of all while statements. Put this list into a results hashmap:
    a. Call PKB API GetStatementType("while"), receives ["3", "6", "10"]
    b. Put list into results map: { "w" : ["3", "6", "10"] }
3. For each clause, evaluate individually. Depending on direction and relationship type (Follows, etc), call the corresponding PKB API. If only one synonym in the clause, e.g. Follows(s, 7) or Follows(6, s), it returns a list of possible statements / variables for that synonym. Then, get a list of all statements of that type (s is a stmt type, according to attributes map). Get the common elements between these 2 lists, and put this list into a hash map. If there are more than one synonym in the clause, e.g. Follows (s1, s2), the additional initial step is to call PKB API to get a list of all possible values for the first argument (s1). Then for each value in this list, do the above and check for possible values of the other synonym (s2). If there exists values for the other synonym, add both s1 and s2 values to two separate lists respectively. After going through all possible "pairings" of s1 and s2, put the two separate lists into the hash map.
    a. Call PKB API GetFollowedBy("7"), receives ["6"]
    b. Call PKB API GetStatementType("while"), receives ["3", "6", "10"]
    c. Get common elements between these 2 lists
    d. Put list into relationships map: { "w" : ["3", "6", "10"] }
4. Filter between the relationships hashmap in step 3 against the results hashmap in step 2. If the same key exists, store only the common elements in the list. If there is no common key, add the key-list entry to the hashmap.
    a. Filter between results map and relationships map, see that same key "w" exists, store common elements
    b. Now, results map: { "w" : ["6"] }
5. Repeat steps 3 and 4 for every clause.
6. After all clauses have been evaluated, check all entries in the results map. If any of the lists are empty, it means that one of the clauses must have evaluated to "false", so the entire query should return None. In this case, return "None" (empty list). If none of the lists in the results map are empty, return the list associated with the to-be-selected entity (key: "s").
    a. No empty lists in results map
    b. To-be-selected entity is "w", look it up in results map
    c. Return ["6"]

5. Testing

Testing is an important part of a software development. Since the team has been split into 3 sub-teams based on components, it is necessary to have integration testing when the new APIs and features are implemented to ensure that the components work together and the API calls work. This happens weekly from week 4 when the component features are pushed to master. Unit testing is dependent on the sub-team, and usually happens at least once a week. This is to test the functionality of the component itself and within subcomponents, for example, QueryEvaluator is calling the correct Query Preprocesser methods etc.

To help us with regression testing from week 4 onwards, we started exploring how Autotester is being used. We figured that there were 2 methods in the TestWrapper.cpp file which we should add in our functions to either parse SIMPLE programs or evaluate the queries. Once the main features to parse and evaluate were up, we have been using Autotester for regression testing.

5.1. Test plan

5.1.1. Inter-Component Testing

Parser

Mini-Iteration #	Testing Activities	Rationale
1 (Week 3)	Unit Testing	Ensure that the parser, controller, validator and logic can interact correctly with each other to provide Follows/Parents functionality. For methods that were not implemented, default values were provided.
2 (Week 4)	Unit Testing Regression Testing	Replaced default values of method implementations to support Uses/Modifies/Pattern functionality while making sure components interacted correctly.
3 (Week 5)	Unit Testing Integration Testing Regression Testing	Fine-tuned previous implementations to consultation feedback and bugs found from unit testing and ran AutoTester to check integration between Parser and PKB.

PKB

Mini-Iteration #	Testing Activities	Rationale
1 (Week 3)	Unit Testing	Ensure API calls to data structures were functioning appropriately for Follows/Parent functions.
2 (Week 4)	Unit Testing Regression Testing	Ensured API calls for Uses/Modifies/Pattern were working as intended. Check for bugs in string converter algorithm. Fixed implementation errors in data structures (support handling of duplicate statements)
3 (Week 5)	Unit Testing Regression Testing	Fix implementation errors.

Query Processor

Mini-Iteration #	Testing Activities	Rationale
1 (Week 3)	Unit Testing	• Ensures that the Query Preprocessor correctly identifies invalid queries. • Ensures that the Query Preprocessor correctly extracts the relevant query details and stores them into a PQLQuery Object. • Ensures that the Query Evaluator correctly filters between the results of different clauses
2 (Week 4)	Unit Testing Regression Testing Integration Testing	• Checks that the Query Preprocessor correctly captures and delivers relevant and important details to the Evaluator, even with added functionalities. • Checks that the Query Evaluator correctly retrieves information from the PKB via the PKB API, for all types of valid queries. • Fix implementation errors.
3 (Week 5)	Unit Testing Regression Testing Integration Testing	

5.1.2. System Testing

Mini-Iteration #	Testing Activities	Rationale
Week 6	Regression Testing System Testing	Parser • Ensure different nesting levels of container statements (if, while, if-while, while-if, etc). • Varying positions of the nested conditional (at the start, in the middle, before the loop ends). • Having similar looking patterns but with different AST structure. • Having nested conditional expressions. Queries • Select synonyms based on attributes declared. • No clause. • Test individual relationships in one such that clause. • Test different types of Pattern clauses. • Test combinations of such that and pattern clauses. • Test usage of common synonyms, and without synonyms. • Test if program correctly detects invalid queries.

This section contains a more detailed explanation of test cases designed for system testing. Towards the end of week 6, the team's testing IC has come up with a concrete plan for testing. After making sure that all components and its features were done and checked with unit and integration testing, we had a meeting to discuss about system testing. All test-cases were run through the AutoTester.

To test the basic functionality of our SPA, we first ran the AutoTester provided to see if our implementation was correct. Following which, we designed our own test cases to test the advanced functionality of our SPA. Our test cases were designed to test functionality (single clauses, multiple clauses etc.), edge-cases and positive/negative cases. We also performed stress testing on our parser.

We divided the system testing into two sections. We first started by testing the functionality of the parser by designing test-cases of SIMPLE source code following these categories below before feeding it trivial PQL queries.

Parser Positive Category:

- Empty stmtList procedure
- Empty stmtList if / while
- Statement blocks
    - Level of nesting
        - 2 Nested
            - While-while
            - While-if
            - If-while
            - If-if
        - Deeper levels more than 2
    - Varying statements before and after statement blocks
        - Position of nested container
        - If … { statements if/while statements }
        - If () { if statements}
        - If () {statements if }}
- Varying amount of white spaces e.g. correct program but everything on one line
    - No white space
    - A lot extra white space
    - White space between the 'tokens'

Parser Negative Category:

- Invalid variable / procedure names
    - No variable name
    - Variable names with invalid keywords
    - White space inside e.g. x 5 = hello;
- If / while block with invalid expressions
    - Missing brackets
    - Extra brackets
    - Invalid syntax before / after the expressions e.g. if ishouldnotbehere (x = 5) test{}

- Statement outside of statement list
    - E.g. x = 5;
    - Proc {....} x = hello;
- Invalid keywords spelt wrongly
    - Procedure
    - While
    - If Print
    - Read
    - Call
- Extra tokens detected
    - E.g. procedure hello moshi moshi { …. }
- Invalid expressions
    - Extra characters present x = x_5; or x = x ++5;
    - Extra white space X = 5 4;
- Empty program

Following which, we tested advanced functionality of the query evaluator first fed correct and trivial SIMPLE source code to our Parser. Invalid queries within clauses are labelled as invalid but within the clauses on documentation, but we have compiled all invalid queries into a separate test case file. Wrong cases are when the queries are valid but are wrong due to other reasons, specific to the clause. We designed query test cases based on the following categories:

- Select with no clause
- Select with 1 clause
    - Modifies
        - 2nd param unknown
        - 1st param: assign | read | procedure | while | if | call | statement
            - Can be known or unknown, e.g. synonym or "a"/7
        - Invalid cases:
            - 1st param is unknown
            - 1st param is print
            - 2nd param is a synonym other than variable
    - Uses
        - 2nd param unknown
        - 1st param: assign | print | procedure | while | if | call | statement
            - Can be known or unknown, e.g. synonym or "a"/7
        - Invalid cases:
            - 1st param is unknown
            - 1st param is read
            - 2nd param is a synonym other than variable
    - Follows(*)
        - 1st param _, 2nd param _, both param _, neither param _
        - Wrong cases:
            - Statement number out of index (negative number or > no. of lines)
            - Statement s1 > statement s2

        - 1st and 2nd param differ in nesting
    - Invalid Cases:
        - var_name or proc_name is passed as parameter
            - Follows*(“x”, 3)
  - Parent(*)
    - 1st param _, 2nd param _, both param _, neither param _
    - 1st and 2nd param differ in nesting
    - Wrong cases:
        - Statement number out of index (negative number or > no. of lines)
        - Statement s1 > statement s2
    - Invalid Cases:
        - A var_name or proc_name is passed as parameter
            - Parent(4, “x”)
  - Pattern
    - Both param _, either param _, neither param _
    - Partial expression and complete expression
    - Wrong cases:
        - Variable does not exist in the source
    - Invalid cases:
        - Assignment attribute is not declared or Entity is not assignment
        - Pass in any parameter in the LHS that is not variable
        - Incomplete or mismatched quotes or underscores on the LHS
            - _”expression” or “expression”_
- Combination of 1 such that and 1 pattern clause
  - 5C2 * 2! = 20 different combinations
    - Modifies + Pattern/Uses/Follows/Parent
    - Uses + Pattern/Follows/Parent/Modifies
    - Follows + Pattern/Uses/Paren/Modifies
    - Parent + Pattern/Uses/Modifies/Parent
    - Pattern + Uses/Follows/Parent/Modifies
  - For the big source, should have 10 combinations
- Invalid queries
  - Syntactically incorrect
    - Wrong number of parameters
    - ‘;’ at end of query
    - Incorrect case for the clauses, e.g. “follows” instead of “Follows”
    - Misspelling of keywords
  - Incorrect parameter types for the clauses
    - These are addressed within their respective clauses above.
  - Using attributes that were not declared in the clauses’ parameters or in the select attribute.

5.2. Examples of test cases of different categories

5.2.1. Unit Testing

SPA Front-end

Sample 1 Test Purpose: Testing the functionality of the validator for relation expressions for conditional statements.

Required Test Inputs: Testing functionality of Validator. Requires varying operators in conditionals.

Expected Test Results: Conditional operators that are invalid such as (=, !, <>, etc) will fail the evaluator. Conditional operators that follow the SIMPLE syntax return true.

Sample 2 Test Purpose: Testing the functionality of the validator for NAMEs.

Required Test Inputs: Testing functionality of Validator. Testing for valid syntax that follows SIMPLE requirement, varying the capitalization of the letters, invalid syntax, e.g. string starts with a number or symbol character, string contains symbol characters.

Expected Test Results: Strings that fit the SIMPLE syntax LETTER(LETTER|DIGIT) + return true otherwise, the validator evaluates the name to be invalid.

Query Processor

Test 1: Attribute Declaration

Test Purpose: Testing the parsing of a simple query, with emphasis on attribute declaration. The attributes *unordered_map* should be created to hold all the attributes declared regardless of Entity and spaces. Select attribute and its entity should also be stored well.

Required test input: Just the ParseQuery() method of the QueryPreprocessor. A query with varying entities (shown below) and multiple attributes from the same entity:

```
ParseQuery("procedure  p;  stmt s1,  s2, s3; assign  as , ass; while www ;  if ifs ; print pnl; read reS; Select s2")
```

Expected Test Results: Valid query check should return true. Select attribute is "s2" and entity is STATEMENT. Attribute map created from the parsing should be equal to the self-written map.

Test 2: Follows Clause (This test is written for all relationships in such that clause as well as pattern clause)

Test Purpose: Ensure that the clause is initialised correctly, and that all variables are stored correctly in the clause. Check whether the attribute related methods in the PQLQuery class can be accessed with the clause object that was just created.

Required Test Input: Just the parseQuery() method of the QueryPreprocessor. A PQL query with one such that clause using the Follows relationship:

```
ParseQuery("stmt s; Select   s such that Follows(s, 4)")
```

Expected Test Results: Valid query check should return true. Select attribute is “s” and entity is STATEMENT. The Clause object retrieved from PQLQuery::GetClauses() should have the left parameter be “s” and the right parameter should be “4”. Clause relationship is FOLLOWS. pql.IsRightAttribute(cFollows) should return false, pql.IsLeftAttribute(cFollows) should return true, and pql.GetLeftAttributeEntity(cFollows) should return STATEMENT.

5.2.2. System Testing

Test 1: Testing Parser’s ability to handle multiple nesting

Test Purpose: Test the ability of the parser to handle a large SIMPLE source program with multiple nesting.

Test case :

<u>Simple Source Program</u>

```
procedure ifInsideIf {
 a = 1;
 if (b == 2) then {
   c = 3;
   if (d == 4) then {
     e = 5;
   } else {
     f = 6;

   }
   g = 7;
 } else {
   h = 8;

 }
 i = 9;
}
```

<u>Query</u>

```
stmt s;
Select s such that Parent*(2, s)
```

Expected Test Results: Parser should not fail when parsing. Query should be valid and return 3,4,5,6,7,8.

Test 2: Test Query Processor’s ability to retrieve pattern queries padded with whitespaces.

Test Purpose: Test the ability of the Query processor to retrieve Pattern queries with padded zeros in expression.

Test Case:

<u>Simple Source Program</u>

```
procedure main {
 x = x + 001;
 y = x*y + y*z;
 z = x*y*z+x*x*y;
 x = (x+1) * 2;
 y = ((x+1)+2)*3;
 z = 3*((x+3) + (y+5));

 if (x1 == x2) then {
   read x;
 } else {
   while (x3 == x4) {
     print x;
   }
 }
 p = 008 + q;
}
```

Query

```
assign a;
Select a pattern a(_,_"08"_)
```

Expected Test Results: Parser should have no problem parsing. Query should be valid and return 11.

6. Discussion

By adhering to the iterative SDLC, the group was able to pace the development of the project and familiarizing ourselves with the new language and unfamiliar platforms while learning about the requirements. The mini-iterations has helped the team to have a goal to work towards every week. This allows features to be implemented progressively, as well as keeping on task of our pace. The good thing was that mini-iteration goals coincide for every sub-team, so we were able to access the relevant APIs across components even if we were working on a different feature. We will keep this up for the future iterations.

Communication was key, and our team demonstrated good communication. Everyone kept each other updated on any progress. Whenever someone faces trouble in their component, he will sound out, and another teammate will lend a helping hand. This shows our teamwork not only within sub-team, but as a whole.

Reflections on Parser implementation

By separating the concerns for our Parser component to include a controller, logic and validator, it was easier to implement the each sub-component as a pair as we did not need to know how the other sub-components operated and could focus on implementing the sub-components correctly.

We experienced difficulties in identifying the more complicated cases for the source syntax and SIMPLE syntax and had to change our implementation a few times to accommodate the newly discovered cases. Thinking back, it would have been easier to first identify and clarify with the teaching all possible syntax before beginning our implementation.

Communicating with the PKB team about the APIs we needed also made designing how we were going to store the design abstractions easier. We had a clear concept of what design abstractions we needed to extract and how to track the relationships between procedures and statements.

Reflections on PKB implementation

One difficulty the PKB team faced was having to constantly add, remove and change API functions. Before we began implementation of the SPA, we sat through a couple of meetings to discuss across sub-teams what API calls were required. We did this by running through basic relationships in SPA (e.g. Parent, Pattern etc.) and deciding what was required from there. Following that, we only met once a week to discuss progress of our implementation and to do testing together.

As we did not run through extensive test cases at the start, the Query Processor team realized that there were many more API calls that had to be implemented to check for more relationships. This ranged from creating trivial API like calls to get the last statement number, to higher level API calls like getting multiple variants from a Pattern query (getting statements or variables that match a pattern).

What made this even more challenging was that there wasn't much efficient two-way communication as most requests for changes were made through Telegram messages out of official meeting times which made it harder for them to communicate what they needed. Thankfully, this issue of incomplete and incorrect APIs was resolved during the last two weeks (week 5 & 6) where we discussed the exact API calls required during our test meetings.

In retrospect, we feel that this could have been avoided (or at least reduced in severity) if we ran through extensive test-cases and use cases before implementation. By creating more advanced test cases and running through them early on, we would have possibly identified certain API calls we would have otherwise missed out. Also, one idea we would possibly explore is having smaller sub-team meetings between Parser and PKB, as well as Query Processor and PKB. These meetings will be solely for discussing any changes in requirements for API calls as bulk of PKB's work is in implementing them.

However, we noted that even if were to make the changes above, we could possibly still miss out on some API calls until we perform testing. As such, our team felt that our decision to complete implementation early in the first two weeks was good, as it gave us buffer time to identify certain gaps in our implementation and API calls. We will be sure to plan similarly for future iterations.

Reflections on Query Processor implementation

The Preprocessor sub-component of the Query Processor is responsible for parsing and passing relevant information to the Evaluator which is important for the evaluation of the query results. Before designing or implementing the sub-component, we spent some time understanding the objectives of both the preprocessor and the evaluator, and how the two should interact. After discussing the requirements of both sub-components together with my sub-teammate, I started designing a class diagram of how objects should be interacted in the Preprocessor, and also the methods that will help the evaluator with information retrieval and checks for the query object. This thorough communication phase before implementation helped the Query Processor team to better understand each other's requirements, and there were less hiccups with regards to method calling and the structure of the Preprocessor.

During the implementation of the Pattern clause, which was designed to be inherited from the parent "Clause" (such that), I realised that c++'s vector, and generally most of their data structures do not allow storing of inherited classes. Initial implementation to store the list of clauses was: vector<Clause>, but it causes object slicing for the Pattern child object that is stored in the vector. After spending some time to read up on this occurrence as well as alternative solutions, we decided to use c++'s smart pointer, unique_ptr, so that information and methods in the Pattern clause will not be sliced. Because of this change of implementation, all methods that took in a clause object as parameter was changed to passing in a pointer. However, not everything was smooth sailing, and we were faced with errors related to the use of unique_ptr. I consulted the team after being stuck for a few days and my peers helped with debugging, where the issue lied in the evaluator creating a copy of the PQLQuery object which contains the said vector. It was because of this incident that made me feel the importance of teamwork. Regardless of the difficulty you face, and regardless of the component you are working on, always seek help and update the team on your progress, so they can help you when needed. This will also help reduce backlog, and we should definitely keep up this practice of helping one another in the future iterations.

As for the implementation of the Query Evaluator, the main challenge was to design a logical process that could accurately record and keep track of the relationships between attributes, attribute values as well as the clauses.

A previous implementation of the Query Evaluator (as described in Section 4.3) was the first adaptation of the logical process. It was designed to consider each clause individually and independently of the other clauses. The reason for this design decision was to reduce coupling between clauses. This was also done to reduce the possibility of errors in the evaluation of one clause compounding to the result of evaluation of the other clauses. From a developer's point of view, this manner of abstraction not only improved readability of code, but also made it easier to track errors in the logic in the program logs.

However, as mentioned in Section 4.3, the above implementation had a significant flaw – it represented the results in terms of a list of possible values for each attribute. This was problematic, as the relationships between attributes were not captured in such a representation. The consequence was as such: evaluating later clauses that invalidated values for a certain attribute from an earlier clause would not invalidate the values of another attribute that is associated with it. To further understand this implementation flaw, please refer to Section 4.3, where it is described in greater detail.

The eventual design used (at least for Iteration 1) is a set of maps, as maps would capture the "association" between the values of different attributes. With this implementation that successfully keeps track of attribute relationships, we were able to cover all types of clause combinations in Iteration 1.

7. Documentation of abstract APIs

The following section consists API calls from the PKB.

7.1. VarTable

- void InsertVariable(VARIABLE var)
    - Inserts a VARIABLE into the PKB.
    - @params var: variable to insert
- List<VARIABLE> GetAllVariables()
    - Returns a list of all VARIABLE in the SPA program.

7.2. ProcTable

- void InsertProcedure(PROCEDURE proc)
    - Inserts a PROCEDURE into the PKB.
    - @param proc: procedure to insert
- List<PROCEDURE> GetAllProcedures()
    - Returns a list of all PROCEDURE in the SPA program.

7.3. Follows, Follows*

- List<STMT_NUM> GetFollows(STMT_NUM followedBy)
    - Takes in STMT_NUM followedBy and returns a single element list containing the STMT_NUM of statement that immediately follows followedBy.
    - @param followedBy: statement number to check
- List< STMT_NUM > GetFollowedBy(STMT_NUM follows)
    - Takes in STMT_NUM follows and returns a single element list containing the STMT_NUM of statement that is immediately followed by follows.
    - @param follows: Statement number to check
- List< STMT_NUM > GetAllFollows(STMT_NUM followedBy)
    - Takes in STMT_NUM followedBy and returns a list of statements that contains all STMT_NUM **follows** that satisfies Follows*(followedBy, **follows)** relationship.
    - @param followedBy: Statement number to check
- List< STMT_NUM > GetAllFollowedBy(STMT_NUM follows)
    - Takes in STMT_NUM follows and returns a list of statements that contains all STMT_NUM **followedBy** that satisfies the Follows*(**followedBy,** follows) relationship.
    - @param follows: Statement number to check

7.4. Parent, Parent*

- void SetParent(STMT_NUM parent, STMT_NUM child, bool isElseBlock)
    - Establishes Parent relationship between parent and child in the PKB.
    - Done by inserting a pair value <parent, child> into followsTable and inserting the value <child, parent> into followedByTable.
    - @param parent: Parent statement number
    - @param child: child Statement number
    - @param isElseBlock: Indicates whether child is in an else block

- List< STMT_NUM > GetParent(STMT_NUM child)
  - Takes in a STMT_NUM child and returns a single element list containing the STMT_NUM of the parent.
  - @param child: Child statement number
- List< STMT_NUM > GetChildren(STMT_NUM parent)
  - Takes in a STMT_NUM parent and returns a list of STMT_NUM **child** that satisfies Parent(parent, **child**).
  - @param parent: Parent statement number
- List< STMT_NUM > GetAllParents(STMT_NUM child)
  - Takes in a STMT_NUM child and returns a list of STMT_NUM **parent** of statements that satisfies Parent*(**parent**, child) relationship.
  - @param parent: Parent statement number
- List< STMT_NUM> GetChildrenOfChildren(STMT_NUM parent)
  - Takes in a STMT_NUM parent and returns a list of STMT_NUM **child** of statements that satisfies Parent*(parent**, child**) relationship.
  - @param parent: Parent statement number

7.5. Modifies for Assignment Statements

- void SetModifies(STMT_NUM statement, VARIABLE variable)
  - Establishes Modifies relationship between statement and variable in the PKB, including all parent container statements and containing procedure.
  - Done by inserting the variable into the set for the statement in modifiesTable and inserting the statement into the set for the variable in modifiedByTable, and recursing the parent of the statement until the containing procedure is reached.
  - @param statement: Statement number
  - @param variable: Variable name modified by statement
- List<VARIABLE> GetModifiesVariables(MODIFIER modifier)
  - Takes in a MODIFIER (STMT_NUM or PROCEDURE) statement number or procedure and returns a list of VARIABLE variables that the statement number or procedure modifies.
  - @param modifier: Statement number or procedure to get modified variables
- List<MODIFIER> GetModifiedByStatements(VARIABLE variable)
  - Takes in a VARIABLE variable and returns a list of MODIFIER (STMT_NUM or PROCEDURE) statement numbers and procedures that modify the variable.
  - @param variable: Variable to get modifier statements or procedures

7.6. Uses for Assignment Statements

- void SetUses(STMT_NUM statement, VARIABLE variable)
  - Establishes Uses relationship between statement and variable in the PKB, including all parent container statements and containing procedure.
  - Done by inserting the variable into the set for the statement in usesTable and inserting the statement into the set for the variable in usedByTable, and recursing the parent of the statement until the containing procedure is reached.
  - @param statement: Statement number
  - @param variable: Variable name used by statement

- List<VARIABLE> GetUsesVariables(USER user)
  - Takes in a USER (STMT_NUM or PROCEDURE) statement number or procedure and returns a list of VARIABLE variables that the statement number or procedure uses.
  - @param user: Statement number or procedure to get used variables
- List<USER> GetUsedByStatements(VARIABLE variable)
  - Takes in a VARIABLE variable and returns a list of USER (STMT_NUM or PROCEDURE) statement numbers and procedures that use the variable.
  - @param variable: Variable to get user statements or procedures

7.7. PatternTable

- List< STMT_NUM > GetPatternStatements(VARIABLE lhs, String rhs, bool isPartial)
  - Takes in 3 parameters, VARIABLE var, String pattern and bool isPartial. Returns a list of STMT_NUM that satisfies the pattern query. If isPartial is true, check for partial match of input pattern, else it must be an exact match.
  - @param lhs: Variable on left side of assignment. If lhs variable in query is _, input should be "_"
  - @param rhs: Pattern expression on right side of assignment. If rhs variable in query is _, input should be empty string ""
  - @param isPartial: Indicates whether we are checking for partial match or exact match of rhs
- List<VARIABLE> GetPatternVariables(String pattern, bool isPartial)
  - Takes in 2 parameters, String pattern and bool isPartial. Returns a list of VARIABLE that are modified by assignments satisfying the pattern query. If isPartial is true, check for partial match of input pattern, else it must be an exact match.
  - @param rhs: Pattern expression on right side of assignment. If rhs variable in query is _, input should be empty string ""
  - @param isPartial: Indicates whether we are checking for partial match or exact match of rhs