CS3203 Software Engineering Project

AY20/21 Semester 2

Iteration 3 Project Report

Team 21

Team Members	Matriculation No.	Email
Choo Qi Le Aaron	A0183581Y	aaroncql@u.nus.edu
Jolyn Tan Si Qi	A0188200M	jolyn.tan.sq@u.nus.edu
Li Zi Ying	A0189497B	ziyingli@u.nus.edu
Lim Yu Rong, Samuel	A0183921A	limyr.samuel@u.nus.edu
Nham Quoc Hung	A0187652U	e0323236@u.nus.edu
Pang Jia Da	A0184458R	pangjiada@u.nus.edu

Consultation Hours: Tuesday 4pm - 5pm

Tutor(s): Yik Ren Jun, Gabriel

Table of Contents

1 Scope

1.1 Source Code

The Static Program Analyzer (SPA) allows for the parsing and validation of SIMPLE source code as per the concrete grammar rules for SIMPLE, specifically:

- Tokenizing of lexical tokens such as *names* and *constants* and other reserved keywords within SIMPLE
- The use of multiple *procedures*, with each containing a statement list with at least one statement
- The handling of non container statements like *read*, *print*, *assign,* and *call* statements
- The handling of container statements like *if* and *while*, with each containing two and one statement list respectively, with at least one statement

An important constraint within this program is that *constants* are assumed to be less than or equal to 2^{31} - 1, the maximum size of a 32 bit signed integer. Any constants used that are larger than this number will throw an unhandled error.

1.2 Program Query Language (PQL)

The SPA allows users to query information about a given SIMPLE source code using the grammar rules of the PQL. It will then return a list of strings representing either statement numbers or variable/procedure names or both. Users can query about certain relationships with three types of clauses: *such that, pattern* and *with*. These clauses operate on integers, strings and a set of 11 Design Entities (*stmt, read, print, call, while, if, assign, variable, constant, procedure, prog_line).*

'Such that' clauses have the following Design Abstractions, which describes a relationship between two parameters:

- *Follows* and *Follows** (F/F*), which describe a pair of statements, with one directly or indirectly following the other
- *Parent* and *Parent** (P/P*), which describe a pair of statements, with one being directly or indirectly nested inside the other
- *Uses* (U), which describes a variable being used by some statement (inclusive of container statements) or procedure
- *Modifies* (M), which describes a variable being modified by some statement (inclusive of container statements) or procedure
- *Calls* and *Calls** (C/C*), which describe a pair of procedures, with one being directly or indirectly called by the other
- *Next* and *Next** (N/N*), which describe a pair of program lines in the same procedure, with one directly or indirectly executing right after the other
- *Affects* and *Affects** (A/A*), which describe a pair of assignment statements in the same procedure, with one directly or indirectly affecting the value of a variable that is used by the other

The following two design abstractions are implemented as possible extensions to the SPA:

- *NextBip* and *NextBip** (NB/NB*), which describe a pair of program lines across procedures, with one directly or indirectly executing right after the other
- *AffectsBip* and *AffectsBip** (AB/AB*), which describe a pair of assignment statements across procedures, with one directly or indirectly affecting the value of a variable that is used by the other

'Pattern' clauses have the following options:

- *assign* clauses, which can highlight specific code patterns used in the right-hand-side of assignment statements, or variables used in the left-hand-side of assignment statements

- *if* and *while* clauses, which can highlight specific variables used in the conditional expressions of ‘if’ and ‘while’ statements

With clauses have the following options:

- *procName* and *varName*: Attributes that can identify the specific name of a procedure or variable used in a statement
- *value*: An attribute that identifies the value of a constant
- *stmt#*: An attribute that identifies the statement number of a given Design Entity

The PQL syntax supports the following grammar in this SPA:

- One or more *declarations* of synonyms to their respective design entity types
- A *Select* clause which allows the selection of the boolean result of the query, a design entity, an attribute of a design entity of a tuple of design entities and/or attributes
- Any number of *‘such that’* clauses, which restrict query results based on their fulfillment of a specific condition in the form of a design abstraction
- The use of integers, strings, synonyms, or wildcards in ‘such that’ clauses
- Any number of *‘pattern’* clauses, which restrict query results based on their fulfillment of a code pattern
- The options of matching or searching for code patterns in ‘pattern’ clauses
- Any number of ‘*with*’ clauses, which restrict query results based on their fulfillment of a specific condition in the form of attributes
- The use of synonyms or synonym attributes in ‘with’ clauses
- The use of the *‘and’* keyword to connect two clauses of the same type

2 Development Plan

2.1 Software Development Life Cycle

The team has decided to adopt a breadth-first iterative SDLC for this project. The breadth-first iterative model allows each member to incrementally work on each of the features, at the same time ensuring that all components in the SPA are well integrated with each other.

The team has decided to divide each iteration into sprints that span one week, with two standups in each sprint. The first standup of the week serves as an update on the completion status of the previous sprint as well as the tasks that the team will work on for the next sprint. The second standup of the week mainly serves as a session for team members to flag any major issues with their component and seek advice for any problems they have encountered. This also encourages team members to actively review each other's code and understand the other components in the SPA.

2.2 Project Management

The team has decided to make use of GitHub issues to manage each sprint across all iterations. At the beginning of each sprint, team members will open up issues for the tasks that they are to complete in the sprint. Members will make micro pull requests that are linked to their relevant issues that they are to complete. This allows for greater integration between coding and the tasks each person has for the sprint, and it also allows for other team members to easily keep track of the progress of everyone.

GitHub's Milestone feature will also allow for a clearer picture of the goals that are set for each iteration, and of the completion status of the iteration.

2.3 Project Plan

Week	Focus of the week
2	Research on relevant algorithms to be used and handle all miscellaneous and administrative tasks (setting up of communication channels, coding environment etc.)
3	Design each component and draft APIs. To start on the most basic viable products -- basic select queries, and handling read and print statements in source code
4	Evaluation of one clause query with focus on Follows/Follows* in the entire SPA. Start on integration tests and autotester tests. Parsing of assignment statements to be done in preparation for evaluation of pattern clauses.
5	Evaluation of two clause queries with one such that clause and one pattern clause in the entire SPA. Population of PKB with Parent/Parent* relationships.
6	Ensure that all Iteration 1 requirements are met (UsesS, ModifiesS). Comprehensive autotester test cases to be created (3 test cases).
Recess	Writing of report for Iteration 1, and completion of any refactoring processes and test debt.
7	Implementation of UsesP/ModifiesP, and implementation of BOOLEAN selection and if/while ‘pattern’ clauses.
8	Implementation of Calls/Calls*, and implementation of ‘with’ clause parsing and evaluation.

Week	Focus of the week
9	Writing of Iteration 2 report, and implementation of tuple/attribute selection. DE and PKB begin groundwork of Next/Next* (e.g. Control-Flow Graph).
10	Full implementation of Next/Next*, and discussion of query optimization.
11	Implementation of NextBip/NextBip*. Implementation of preprocessing clauses in the Query Optimizer.
12	Implementation of AffectsBip/AffectsBip*. Implementation of grouping and merging tables more effectively in the Query Optimizer.
13	Feature freeze. Focus on writing System Test Cases and the report only.

2.4 Iteration Plan

Legend: Each task goes in the format [Name] (Task). The Name tag is optional. If there is no name tag, the person in charge of the component will do it. If the task is meant for everyone, then it will be indicated by [Team] Task.

Names:
[A] Aaron
[JD] Jia Da
[ZY] Zi Ying
[S] Samuel
[J] Jolyn
[H] Hung

2.4.1 Iteration 1 Plan

	Week 2	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Source Processor (Aaron)	Research on parsing algorithms (e.g Recursive Descent, Shunting Yard)	Implement parsing and validation of read and print statements.	Implement parsing and validation of assign statements.	Implement parsing and validation of while statements.	Implement parsing and validation of if statements.

	Week 2	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Design Extractor (Jia Da and Zi Ying)	Discuss efficient ways to store PKB information to make extraction of Design Abstractions easier	Research on algorithms that deal with left recursion for AST. Store read and print information into PKB.	Population of PKB with Follows and Follows* relationships.	Population of PKB with Parent and Parent* relationships.	Population of PKB with Uses and Modifies relationships.
PKB (Hung)		Design the PKB API and set up tables to store all design entities and allowing QE to retrieve design entities.	Setting up tables for Follows and Follows* and expose relevant APIs for these tables.	Setting up tables for Parent and Parent* and expose relevant APIs for these tables.	Setting up tables for Modifies and Uses and exposing relevant APIs for these tables.
Query Evaluator (Jolyn)		Evaluation of select queries with no clauses. Ensure integration with PKB and Query Parser.	Implement clauses and the evaluation of multiple clauses with focus on Follows/* queries.	Implement the evaluation of multiple clauses with focus on Parent/* and pattern queries	Extend algorithm to correctly evaluate Uses and Modifies clauses with pattern queries

	Week 2	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Query Parser (Samuel)	Research on parsing algorithms (e.g Shunting yard)	Parsing of select queries with no clauses	Parsing of Follows and Follows* queries.	Parsing of pattern expressions in pattern clauses, parsing of Parent and Parent* queries	Parsing of Modifies and Uses queries
Testing (Aaron)	[Team] Set up IDE and test autotester	[Team] Ensure that unit tests for each component are written extensively.	[A, J] Integration of autotester with the SPA [S, J, H] Integration test for Query Processor and PKB [H, JD, ZY] Integration test for Design Extractor and PKB	[Team] To extend integration tests for each component to cover newly implemented features [A] Invalid source program for Autotester test case	[A, J] Autotester Test Case 1 for source program with complex syntax and simple queries [S, J] Autotester Test Case 2 to verify the correctness of queries with all design abstractions

	Week 2	Week 3	Week 4	Week 5	Week 6
Testing (cont.) (Aaron)					[A, ZY, H] Autotester Test Case 3 to verify correctness of queries with complex queries
Documentation (Zi Ying)	[Team] Decide on coding standard		[S] Report: Scope [J] Report: Development Plan [ZY] Report: Coding and Documentation Standards [H, JD] Report: API Design [A] Report: Testing, SPA design		

	Week 2	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Miscellaneous	[A] Setting up of GitHub issues and labels [Team] Set up Google Drive and Teams	[Team] Prepare for Oral Presentation 1 and clarify any existing doubts about Basic SPA.		[A] Setting up of GitHub Actions for automated tests	[Team] Submission of Iteration 1 code and test cases

2.4.2 Iteration 2 Plan

	Week 7	**Week 8**	**Week 9**
Source Processor (Aaron)	Parsing and validation of multiple procedures and call statements		Creation of the CFG for use to extract design entities for Next/Next*
Design Extractor (Jia Da and Zi Ying)	Population of full assign and container pattern information and population of Calls/Calls*	Full Modifies/Uses relationship information	Population of with clause information and Control Flow Graph creation
PKB (Hung)	Storage for Modifies(p, v)/Uses(p,v) and If/While pattern and expose relevant APIs for these tables	Storage for Calls/Calls* and expose relevant APIs for these tables and for evaluation of 'with' clauses	Storage and APIs for Next/Next*
Query Evaluator (Jolyn)	BOOLEAN and prog line selection, evaluation of UsesP, ModifiesP, Calls/Calls*, evaluation of multiple clauses, table merging correctness	Evaluation of pattern while and ifs, with clauses, tuple selection	Selection of attributes

	Week 7	**Week 8**	**Week 9**
Query Parser (Samuel)	Queries with BOOLEAN, new synonyms (prog_line and call), refactoring of program to improve robustness for long queries	Queries with Calls/Calls*, queries with pattern for ‘if’ and ‘while’, queries with multiple clauses, queries with ‘and’ keyword	Queries ‘with’ clauses and attributes, queries with tuples as selected objects, queries with selection of attributes
Query Optimizer (Aaron/Jolyn/Sam)			Planning of Query Optimization I.e how to sort clauses, how to merge tables in a more efficient manner
Testing (Aaron)	Create python test runner to run and check all autotester tests efficiently	Week 7&8 AutoTester TC (*Calls/Calls*/Modifies/Uses/pattern*)	Create SIMPLE source code generator to test parser and for ease of autotester TC creation [J] Iteration 2 TC2: complex queries [JD, H, ZY] Iteration 2 TC3: complex source code to test extraction of design abstractions

	Week 7	**Week 8**	**Week 9**
Documentation (Zi Ying)		[Team] Work on comments and feedback on report from Iteration 1	[Team] Update report based on newly implemented features [A] System testing test plan [J, S] Extension for Iteration 2
Miscellaneous	[Team] Organise Github issues from Iteration 1 and close milestone. Reorganize issues into Iteration 2 milestone.	[Team] Preparation for OP 2	

2.4.3 Iteration 3 Plan

	Week 10	**Week 11**	**Week 12**	**Week 13 (FEATURE FREEZE)**
Source Processor (Aaron)	Enhance efficiency of lexer to deal with longer source codes	CLI-based Python script to generate random SIMPLE source code for stress and correctness testing	Creation of the CFG explosion for use to extract design entities for AffectsBip/ AffectsBip*	
Design Extractor (Jia Da and Zi Ying)	Population of Next/Next*Affects/Affects* relationship	Population of NextBip/NextBip* relationship	Population of AffectsBip/AffectsBip* relationship	
PKB (Hung)	Storage and APIs for Affects/Affects*	Storage and APIs for NextBip/NextBip*	Storage and APIs for AffectsBip/AffectsBip*	
Query Evaluator (Jolyn	Next/Next*/Affects/ Affects* queries	NextBip/NextBip* queries, and improving efficiency of Query Evaluator	AffectsBip/AffectsBip* queries	

	Week 10	**Week 11**	**Week 12**	**Week 13**
Query Parser (Samuel)	Next/Next*/Affects/ Affects* queries	NextBip/NextBip* queries, and improving efficiency of Query Parser	AffectsBip/AffectsBip* queries	
Query Optimizer (Jolyn)	Implement sorting of clauses based on number of synonyms and removal of repeated clauses	Implementing grouping of ResultTables by synonyms, merge only clauses in related groups	Implement sorting of clauses based on Design Abstraction	
Testing (Aaron)		[J, A] Write Autotester test cases to test effectiveness of query optimization	[S] Creation of invalid queries Autotester test case [J] Creation of complex queries Autotest test case [A] Creation of invalid source code test case	Hard to parse source (nested parentheses, arbitrary white space, keywords and variable / procedure names) Invalid simple source

	Week 10	**Week 11**	**Week 12**	**Week 13**
Testing (cont.) (Aaron)			[H,JD, ZY] Creation of complex DE source programme (deep nesting)	
Documentation (Zi Ying)	[Team] Submission of report			[Team] Writing of Iteration 3 report with new features (Next, Affects, QueryOptimizer)
Miscellaneous	[Team] Organise Github issues from Iteration 2 and close milestone. Reorganize issues into Iteration 3 milestone.		[Team] Prepare for Oral Presentation 3	

3 SPA Design

3.1 Overall Architecture

3.1.1 Overview of the SPA

The main components of SPA closely follow the recommendations given in the GitHub Wiki for CS3203. Figure 3.1 shows the overall architecture of the program:

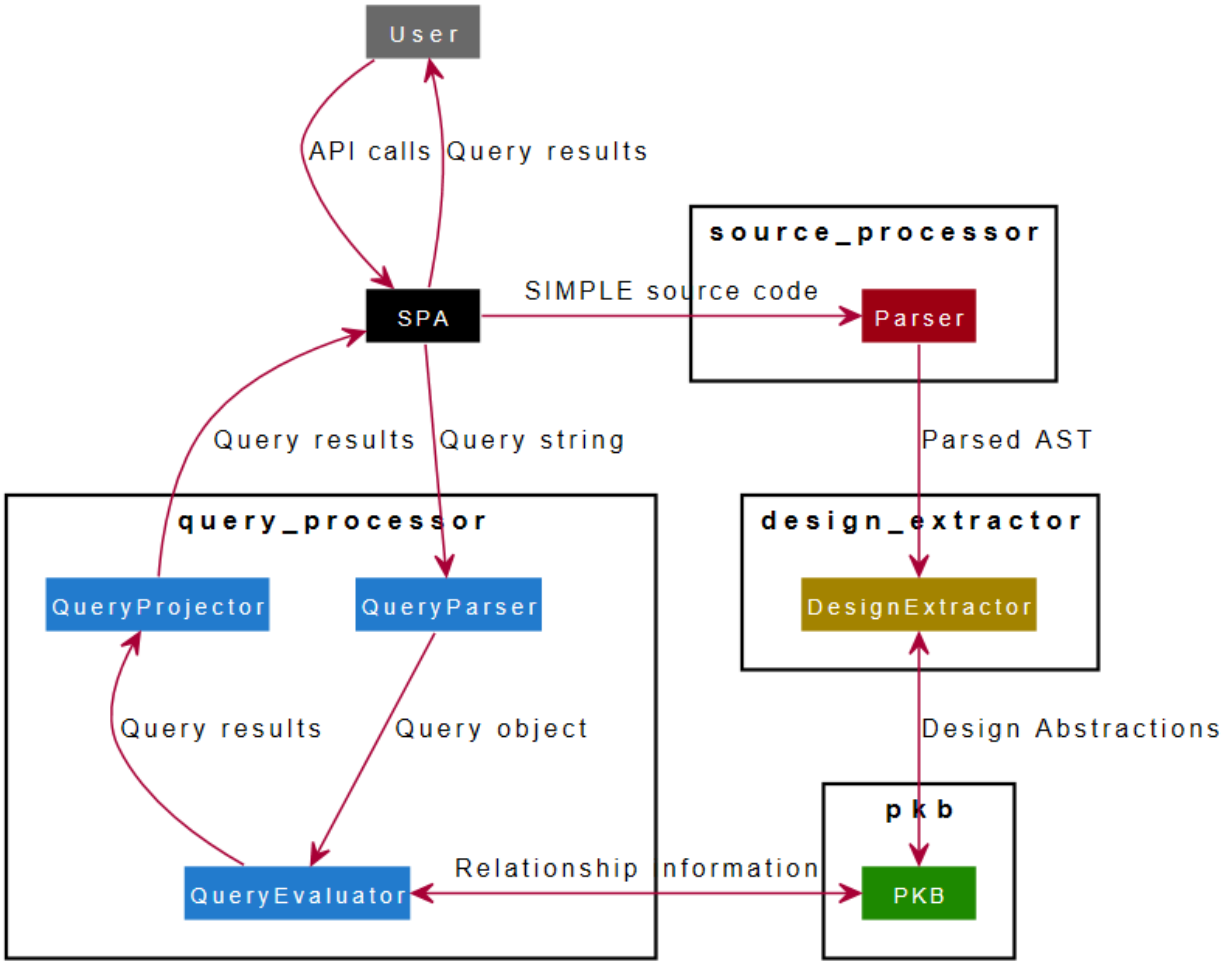

Figure 3.1 SPA Architecture Diagram

Figure 3.1 introduces the colour scheme of components that will be used in the rest of this report. More specifically, the following colours and their related shades represent:

- Red: Source Processor
- Yellow: Design Extractor
- Green: Program Knowledge Base
- Blue: Query Processing Subsystem
- Black: SPA class
- Grey: External agents (e.g. users)

In general, this colouring scheme will stay relevant for other diagrams in this report, unless otherwise stated.

Figure 3.1 is **not** a UML diagram. The data flows represented are abstractions, and may not represent concrete API calls. They are merely meant to give a broad overview on the structure of the program. For more detailed information, refer to the individual design sections of each component.

The arrows in Figure 3.1 describe the main data flows of the components in the program, along with a label describing the information being transferred. For example, the Query Evaluator and the PKB transmit information about relationships (e.g. design abstractions) to each other.

The ‘User’ (colored in grey) in Figure 3.1 represents any entity using this SPA program. They can interact with the SPA program through the API calls implemented in the SPA class, and will receive the results of their queries.

Main Components

- **SPA**: the main entrypoint of the entire program. It is responsible for interfacing with the Source Processor, Design Extractor, and Query Processor. The two exposed methods are responsible for parsing source code and parsing queries, and can be used by a user to execute the program.

- **Parser**: responsible for the parsing and syntax/semantic validation of the SIMPLE source code. Once parsed, it will generate the valid AST for use by DesignExtractor. This Parser class is the main point of entry for all parsing related API, and which contains only static methods (ie. cannot be instantiated).

- **DesignExtractor**: responsible for extracting all the design entities and design abstractions from the given AST and populating the PKB with the relevant data. This class contains only static methods and cannot be instantiated.
- **PKB**: responsible for storage of design entities and design abstractions for efficient retrieval by Query Evaluator. In particular, the PKB provides concrete APIs for the Design Extractor and Query Evaluator in `PKB.h`.
- **Query Parser**: responsible for the parsing and syntactic/semantic validation of the PQL queries. Identifies relevant information from the query string and produces a Query object containing those elements.
- **Query Evaluator**: responsible for the evaluation of a Query object containing clauses. Retrieves relevant information from the PKB and stores intermediary results, finally returning the selected synonym, tuple, attribute or BOOLEAN to the SPA.
- **Query Projector:** responsible for formatting the data result of the query from the Query Evaluator into strings that are displayed by the SPA.

3.1.2 Design Decisions for Overall Architecture

Design Consideration 1: Organization of Source Processor and Design Extractor

Problem Statement: Choice of organization of Source Processor and Design Extractor.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Execution of core parsing and design extraction requirements
The Source Processor and Design Extractor must be able to parse SIMPLE source code, and extract all design abstractions correctly.

Evaluation Criteria:

1. Adherence to design principles
2. Ease of future extensions

Description of Possible Solutions:

Having the Design Extractor component be a part of the Source Processor component
As per the Wiki, the Design Extractor can be subsumed within the Source Processor component.
Having the Design Extractor be a separate component from the Source Processor component
Differs from the design as shown in the Wiki, where the Design Extractor is a separate component (with its own namespace), different from the Source Processor component.

Evaluation of Solutions

Adherence to design principles	
Design Extractor subsumed in Source Processor	Implies that there is a tight coupling or dependency on the Source Processor for the Design Extractor. This generally means that sound design principles like SRP are not strictly followed.
Design Extractor	Implies that the Source Processor is independent from the Design Extractor. This may mean that more sound design principles like SRP

separated from Source Processor	are followed.

Ease of future extensions	
Design Extractor subsumed in Source Processor	Since this implies tight coupling between the Source Processor and Design Extractor, it means that any future extensions will require developers to have sound knowledge of both such components. Tests may also be more tedious since the use of unit tests require stubs for each component.
Design Extractor separated from Source Processor	Since this implies higher cohesion of the Source Processor and Design Extractor components, it means that any future extensions may be easier to write. A feature is likely to be confined to a specific component in the Source Processor and Design Extractor, leading to easier development and testability.

Final Choice

Since SPA should be written for maximum testability and extensibility, where each and every component should be made as independent from one another as possible, it made much better sense for the Source Processor to be as decoupled from the Design Extractor as possible. As discussed in this Source Processor section above, the design of the current architecture is made with such a consideration in mind. Moreover, since neither the Source Processor nor the Design Extractor components reference one another, it does not make sense for the Design Extractor components to be subsumed within the Source Processor component.

Design Consideration 2: using C++ namespaces for each component

Problem Statement: Choice of using C++ namespaces for every component

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Execution of individual components
Each individual component has to be able to execute and perform their functions as per normal, regardless of whether namespaces are used

Evaluation Criteria:

1. Ease of implementation
2. Maintainability

Description of Possible Solutions:

Having one single namespace for the entire SPA
The entire SPA could be grouped under a single namespace
Having each component have its own namespace
Each of the main components: source processor, design extractor, PKB, query processor will have its own namespace

Evaluation of Solutions

Ease of implementation	
Single namespace for whole SPA	Easier to implement as there is only one namespace to consider
One namespace for each component	Slightly harder to implement as there is the inconvenience of remembering to use the fully qualified names

Maintainability	
Single namespace for whole SPA	Hard to maintain, large namespace means that there is a higher chance of polluting the global namespace, especially when a component depends on multiple other components and are included in the file
One namespace for each component	Logical way of having similar sounding functions, variables and/or classes across the different components without clashing or using the wrong functions accidentally

Final Choice

C++ namespaces were chosen for use for each component. The biggest disadvantage of this is the inconvenience of remembering to use the fully qualified names. However, our team has come to the conclusion that this will be much more beneficial in the long run as mentioned above. Since the team is adopting an iterative breadth-first approach to our development, this minimises the headache of trying to coordinate our namings.

3.2 Source Processor

3.2.1 Overview of Source Processor

The Source Processor is responsible for parsing, validating, and creating the resulting Abstract Syntax Tree (AST) given a SIMPLE source code. Due to adhering to the Single Responsibility Principle (SRP), the parser *does not* populate the PKB directly and thus does not depend on the PKB at all. Instead, the Source Processor's only responsibilities are to validate that the SIMPLE source code is semantically valid and to create the AST if so. This AST will then be passed to the DesignExtractor which is the component responsible for extracting all design entities to populate the PKB.

3.2.2 Design of Source Processor

The following diagram shows the class diagram of the most relevant classes used when parsing a SIMPLE source code. Note that to simplify the diagram, transitive relationships are not shown (but which can be taken to be implied).

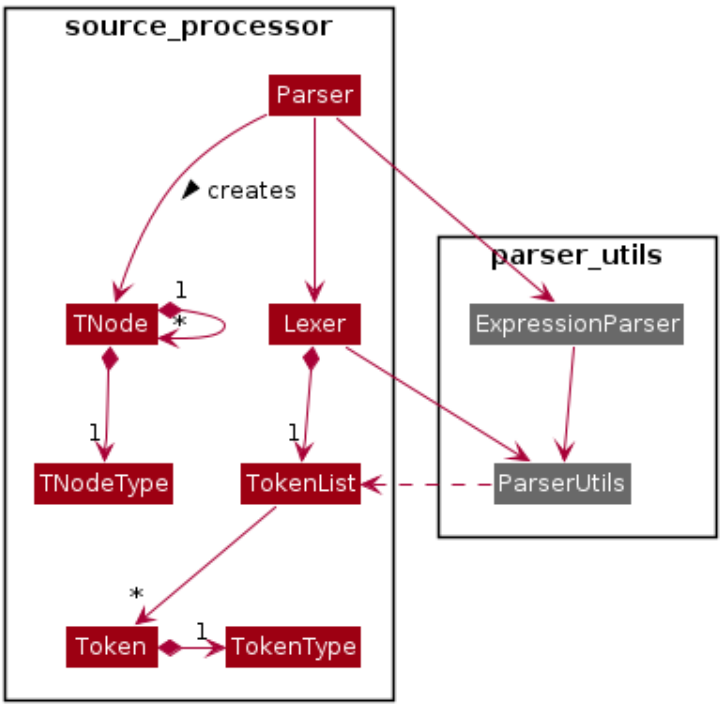

Figure 3.2.2.1 Source Processor Class Diagram

There are 2 main driving parts: the lexer, responsible for extracting and creating tokens from the input SIMPLE source, and the parser, responsible for forming the Abstract Syntax Tree (AST) from the tokens for use by the Design Extractor.

The primary role of the lexer is to classify tokens and make the corresponding Token objects from the input source code. The tokens are extracted and classified using regexes, while disregarding any semantic grammar rules (which will be left for the parser to validate). If the regex is unable to match the current input source to a regex pattern (eg. if an invalid token like “@” is encountered), the lexer will throw a runtime error with a description of the error.

The primary role of the parser is to then take these corresponding tokens, verify that it is semantically sound with SIMPLE’s concrete grammar rules, and then to form the AST that will be used by the Design Extractor. Here, the context-sensitive grammar rules are also further verified. This parser is implemented using the top-down recursive descent algorithm, where the first unexpected token will immediately throw a runtime error.

The AST that is eventually passed to the Design Extractor is represented using TNode (tree node) objects, where each node of the tree is a single TNode object. Each of them contains a vector of children TNode pointers as well as various necessary information (eg. statement number, value, type of node, etc.). As such, the standard breadth or depth first traversals can be used to traverse the AST.

3.2.1.1 Interaction with Design Extractor and PKB

Before diving into how the lexer and parser works, the interaction between the Source Processor, Design Extractor and PKB will first be described. Assume the following SIMPLE source code:

```
procedure main {
  read x;
}
```

The following is the sequence diagram for the Parser, DesignExtractor, and PKB where the AutoTester is called to parse the above source code:

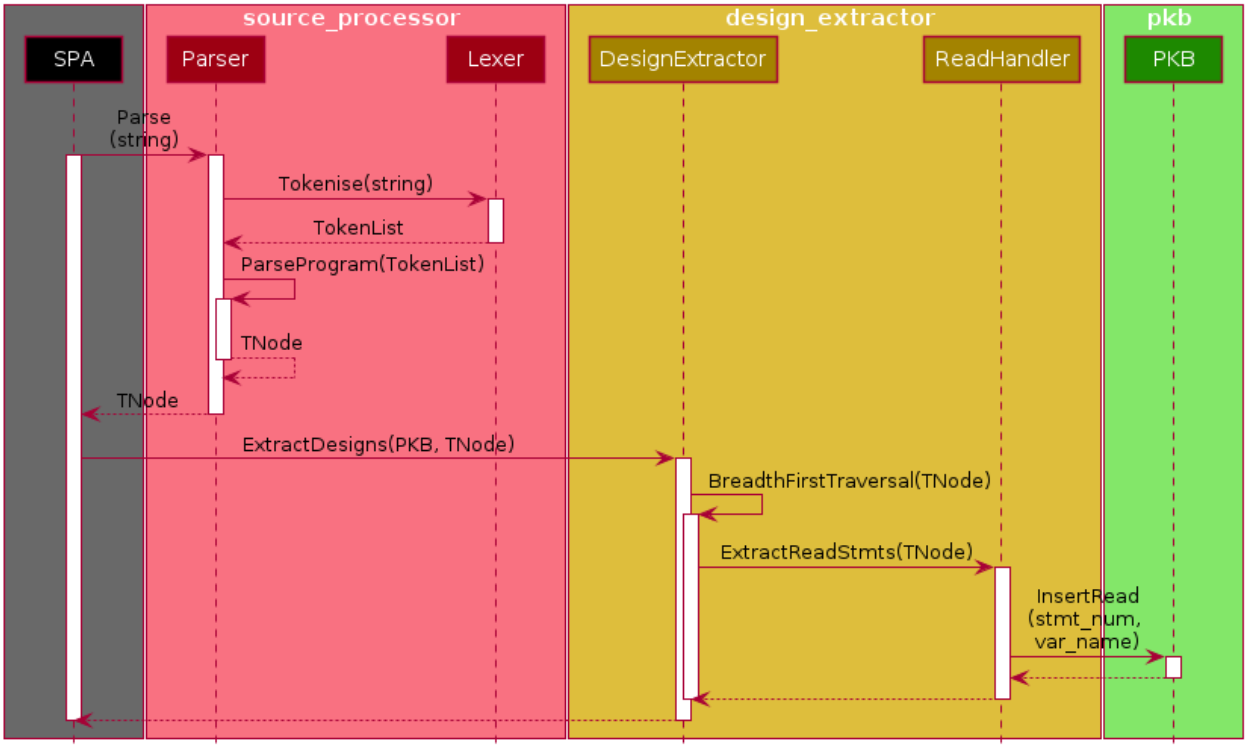

Figure 3.2.1.1 Source Processor to PKB Sequence Diagram

Note that some less significant utility and helper classes are left out in this sequence diagram. Most notably, the DesignExtractor will also call a few other handlers other than just the ReadHandler to extract all the other design entities. These handlers behave similarly to the ReadHandler, inserting the corresponding design entity to the PKB.

3.2.2 SIMPLE Program Lexer

3.2.2.1 Extraction of Tokens from SIMPLE Source Code

To extract the Tokens, the SIMPLE source is matched to a set of regexes which describe all the valid Tokens as per the concrete grammar rules of SIMPLE. The first valid Token is then removed from the SIMPLE source, and the next chunk of raw source code is then recursively called to the same matching algorithm again. This process occurs until the SIMPLE source code is fully consumed. If the regexes are unable to find

a suitable match at any point, then the chunk of raw source that is currently being encountered must have a syntactic error.

As seen in Figure 3.2.2.1, each Token contains a member enum class TokenType, which describes what type of token it is. Upon encountering a token, the Lexer will use this enum class to classify which type of Token it is. For instance, the regex to match to a variable name is as follows:

```
^\\s*([a-zA-Z0-9]+)\\s*([\\s\\S]*)
```

- **^**: ensures that this regex is anchored to the start of the raw SIMPLE source code that is being matched
- **\\s***: ensures that any whitespaces before the variable name is ignored
- **([a-zA-Z0-9]+)**: the capture group for the variable name which must contain at least one of any combination of letter or digit
- **\\s***: ensures that any whitespaces after the variable name is ignore
- **([\\s\\S]*)**: the capture group for the next chunk of raw source to test for

As can be seen, this regex is actually more lax than what the concrete grammar rules allow for (the capture group for the variable name will still classify variable names starting with a number even though it is not a valid variable name). The actual validation occurs when instantiating the Token object, which again uses regex to strictly match the value given to that of the concrete grammar rules.

This was a conscious design decision since the error thrown to the end user can be more informative. If the Lexer is unable to match the current source to any token, it will simply print the current chunk of raw source and alert the user that there exists an error in that current chunk. As such, if the regex used in the lexer strictly only allows for a valid SIMPLE variable name and an invalid variable name like `420blazeit` is used, this chunk of raw source will be unable to match with any regex and the above generic error will be printed. This is not as informative as the current implementation, where the user will be notified of the exact syntactic error in the variable name because the checking is

done within the Token object itself after it has been classified as “most likely” a variable name.

3.2.2.2 Differentiating Reserved Keywords from User Defined Names

In SIMPLE, the reserved keywords like “procedure” and “read” can also be used as a user defined name[1]. Thus, whenever a reserved keyword is encountered, there is a need to decide whether it is really a reserved keyword, or if it is a name. This must occur within the Lexer since classification and validation of tokens is the responsibility of the Lexer and not the parser.

As per the concrete grammar rules, the following heuristic will be able to correctly determine whether a reserved keyword should be classified as a user defined name:

- If the previous token is already a reserved keyword, then the current token (if it can potentially be a reserved keyword) must be classified as a name

As such, whenever a potential reserved keyword token is found, the previous token that was found will then be checked if it was also a reserved keyword. If it is, then the current token will be classified as a name. Else, the current token can safely be classified as a reserved keyword.

This algorithm is valid because all the reserved keywords can only be one of the following:

- Used at the start of a new statement (eg. “read”, “call”, “print”)
- Used when there should not be a variable name before it (eg. places where reserved keywords like “procedure”, “then”, “else” appear in a valid SIMPLE source do not have variable names before it)

[1] Although the actual code implementation classifies variable names and procedure names separately, both of them are collectively referred to as *user defined name* or *name* in this report for brevity

3.2.2.3 Design Decisions for SIMPLE Program Lexer

Design Consideration 1: using enum class vs inheritance/polymorphism

Problem Statement: Choice between using a member variable enum class or polymorphism to differentiate the *TNode* types.

Constraints of the Problem:

The different TNode types can be properly differentiated
The AST structure largely follows the one laid out in the Wiki. As such, there is a need for the DE to differentiate between the different nodes in that tree. For example, some nodes may represent a *read* node, while some may represent a *statement list* node, etc.

Evaluation Criteria:

1. Performance
2. Ease of future extensions
3. Adherence to design principles

Description of Possible Solutions:

Using inheritance/polymorphism to create multiple child classes which extends from a base node clase
There can be a base *TNode* class which multiple child classes like *ReadTNode* can extend from. The type of node can then be determined using runtime type checking.
Using a single *TNode* class which stores an enum class variable
The enum class variable is then used to determine what type of node the *TNode*

object represents during runtime.

Evaluation of Solutions

Performance	
Inheritance	Known to be very inefficient in C++. Requires the use of `dynamic_cast` which is known to be very computationally expensive.
Enum class variable	Very efficient in C++ to check for equality of enum classes. When compiled, an enum class is simply an integer which has the same time complexity of checking for integer equality.

Ease of future extensions	
Inheritance	Requires more boilerplate code to be added whenever a new node type is added. For example, a new *ChildTNode* class will need to be created and then included in every file which requires the use of this new child class.
Enum class variable	Requires minimal code to be added. Any addition or deletion of a node type will simply require a one-liner change in the underlying enum class which stores the node types.

Adherence to design principles	
Inheritance	In a pure OOP setting, this adheres much more strongly to sound OOP principles. Developers who are well versed in OOP will find this approach intuitive and easy to follow.

Enum class variable	Dissimilar in implementation compared to purer OOP languages like Java. Developers familiar with purer OOP languages but not with C++ might have a harder time understanding the code.

Final Choice

Ultimately, the choice of using the enum class was obvious. The developers working within this product are unlikely to be changed, and even for one new to this paradigm of using enum classes instead of true polymorphism, it would also be relatively easy to learn.

Moreover, the type checking of the nodes are done very frequently, both in the SP and the DE. Extensive use of the `dynamic_cast` operator has been seen as a relatively bad practice: both because of its inefficiency, and because the resulting codebase will become much less readable in the long run.

3.2.3 SIMPLE Program Parser

3.2.3.1 Recursive Descent & Shunting Yard Algorithm

Although SIMPLE is a context sensitive language, the Recursive Descent parsing algorithm was still chosen. Simply put, this algorithm will recursively try to pattern match the current token with the SIMPLE grammar rule that is expected. If an unexpected token is encountered, it will exit, showing an appropriate error message to the user.

For example, when parsing a new statement, the parser will first test the first token in that statement to determine which type of statement it is (eg. read, print, assign statement, etc.).

Using the following SIMPLE grammar rule for a read statement:

```
read: 'read' var_name';'
```

The code snippet that will parse and create the corresponding AST for this read statement is as follows:

```
void Parser::ParseReadStatement(TNode* stmt_list_node) {
  // expect the 'read' keyword
  tokens.PopExpect(TokenType::Read);
  // expect the var_name
  Token var_token = tokens.PopExpect(TokenType::VariableName);
  // expect the ';' keyword
  tokens.PopExpect(TokenType::Semicolon);

  // add read_node->var_node branch as child of stmt_list_node
  TNode* read_node = new TNode(TNodeType::Read, statement_number);
  TNode* var_node = new TNode(TNodeType::Variable, statement_number,
                              var_token.GetValue());
  stmt_list_node->AddChild(&read_node->AddChild(var_node));
}
```

- `tokens` is the static TokenList which stores all the tokens to be visited
- The `PopExpect` method of `tokens` will throw if the first token currently in it is not of the correct type
- The ‘read’ keyword, variable name, and ‘;’ keyword must exist and expected to be in this exact order
- The AST for this read statement is then created and added as a child of the statement list that was passed to this function

This algorithm is repeated for all other grammar rules, except for `expr`. When an expression is met, the parser will instead switch to the Shunting Yard algorithm. The Shunting Yard will first produce the corresponding Reverse Polish Notation (RPN) of the expression, which is then used to produce the AST for said expression. The decision to do this is detailed in section 3.2.3.4.

3.2.3.2 Validation of SIMPLE Source Code

The validation algorithm implemented can be broken down into three main chunks:

1. Ensuring that the tokens in the SIMPLE source code can be tokenized and are valid (already discussed in section 3.2.2.1)
2. Ensuring that the SIMPLE source code adheres to the concrete grammar rules (mostly done within the recursive descent algorithm)
3. Ensuring that expressions when parsed using the Shunting Yard algorithm is a valid SIMPLE expression as per the concrete grammar rules
4. Ensuring that call statements call an existing procedure and that no recursive calls are allowed

The third point about validating a SIMPLE expression will first be discussed. Because the Shunting Yard algorithm is more forgiving than what the SIMPLE grammar allows, some checks must be put in place to make sure that the given expressions conform to a valid SIMPLE expression.

For instance, the following expression is clearly invalid:

```
1 + 2 +
```

However, blindly following the Shunting Yard algorithm[2] will still parse this without throwing any errors. Moreover, some valid mathematical expressions using implicit multiplication like `2x` or `2(x)` will also not be detected.

As such, within ParserUtils, two checks are used to detect whether the given expression can be a valid expression (if it also can be successfully parsed by the Shunting Yard algorithm).

Firstly, the `HasAlternatingOperatorPattern` will tackle the first issue by checking that a valid expression meets the following two criterias:

1. Ignoring all parentheses, operators and operands must be alternating
2. First and last token cannot be operators

[2] As given in https://en.wikipedia.org/wiki/Shunting-yard_algorithm

Then, the `HasValidParenthesisPattern` will tackle the implicit multiplication issue by checking that a valid expression meets the following four criterias:

1. Immediately after an operator cannot be a right parenthesis
2. Immediately after an operand cannot be a left parenthesis
3. Immediately after a right parenthesis cannot be an operand or left parenthesis
4. Immediately after a left parenthesis cannot be an operator or right parenthesis

If an expression successfully passes these 2 checks, and can also be parsed successfully by the Shunting Yard algorithm, then the expression must be a valid SIMPLE expression.

Lastly, the Source Processor will also verify that all call statements must call an existing procedure and that the calls are not recursive in nature. In order to verify that all call statements call an existing procedure, during the parsing process, a hashmap is used to map a procedure to every procedure that it calls. At the end of the parsing process, the hashmap is then iterated through, checking that every procedure called also exists in the hashmap itself. If the hashmap does not contain a procedure that was called, it means that the source contains an invalid call statement calling a non-existent procedure.

After the above validation, the hashmap can then be used as an adjacency list: a directed graph where a directed edge exists from procedure *p* to another procedure *q* which was called from *p*. Thus, to validate that no recursive calls were made is akin to checking if said graph is a Directed Acyclic Graph (DAG). This can further be reduced to finding if a back edge exists in the graph. As such, Depth First Search (DFS) will be able to solve this problem efficiently with a linear time complexity of $O(V + E)$, where V is the number of procedures, and E is the number of call statements.

The algorithm to perform such a check is made up of a DFS function (called *dfs*) which returns true if a cycle is found and a *main* function which calls the DFS on every vertex in the graph. Two different sets, *visited* and *visiting* are needed.

The pseudocode for the *main* function is as such:

1. For every vertex *v* in the graph, call *dfs(v)*:
    1.1. If *dfs(v)* returns true, then the graph has cycles

The pseudocode for *dfs* is as such:

1. If *visited* does not contain *v*:
    1.1. Add *v* to both the *visited* and *visiting* set
    1.2. Undergo DFS, and for every vertex *u*:
        1.2.1. If *visited* set does not contain *u* but *dfs(u)* is true, then return *true*
        1.2.2. If *visiting* set contains *u*, then return *true*
2. Remove *v* from *visiting* set
3. Return *false*

Even though the *dfs* function is called for every vertex in the graph, the use of the *visited* set ensures that the multiple runs of *dfs* will not occur if the vertex has already been found to not contain a cycle in earlier/recursive calls to *dfs*. As such, this guarantees a linear runtime as said above to efficiently check if the graph is a DAG.

3.2.3.3 Design Decisions for SIMPLE Program Parser

Design consideration 1: tight coupling with PKB vs loose coupling with PKB

Problem Statement: Choice between allowing the Parser to also extract some design entities while parsing (tight coupling with PKB) or not (loose coupling with PKB)

Constraints of the Problem:

All design entities can be extracted successfully regardless of where it is done
Allowing the parser to extract some design entities or not should not affect whether all design entities of the program can be extracted or not.

Evaluation Criteria:

1. Adherence to design principles
2. Testability
3. Performance

Description of Possible Solutions:

Tight coupling with PKB: extract some basic design entities in initial parse of the source program
For instance, while parsing a statement, the Parser can immediately identify what type of statement it is and then fill up the PKB accordingly.
Loose coupling with PKB: Parser does not extract any design entities nor is it dependent on the PKB at all
The Parser in this case only has a single responsibility: to parse the program successfully and to produce a valid AST.

Evaluation of Solutions

Adherence to design principles	
Tight coupling with PKB	This does not adhere to Single Responsibility Principle (SRP), since the Parser now has more than a single responsibility in addition to its main responsibility of producing a valid AST. This also makes the code base less cohesive in nature since the extraction of design entities is effectively spread across both the SP and the DE.
Loose coupling with PKB	Adheres favourably to SRP and promotes high cohesion in the codebase, where the implementation of a major feature is largely contained within one specific component instead of being spread out.

Testability	
Tight coupling with PKB	Much harder to test for two reasons: 1. For SP's unit test: requires the use of stubs for the PKB 2. For DE's unit test: requires the use of stubs for the SP *and* to also mock/generate the correct basic design entities which the SP should account for
Loose coupling with PKB	Much easier to test. Tests for the Parser can solely focus on the creation of the AST and of the validation of the SIMPLE source code. There is no need to use stubs for the PKB as well in the unit tests.

Performance	
Tight coupling with PKB	May be more efficient, since there is no need for the DE to extract the basic design entities, potentially decreasing the number of passes needed on the AST.
Loose coupling with PKB	May be less efficient, since the DE has to first extract the basic design entities first before the more advanced ones.

Final Choice

The use of loose coupling with the PKB was chosen over the tight coupling since the advantages far outweigh any disadvantages. The time complexity for both of these approaches are asymptotically similar, with a linear *O(n)* runtime with regards to the number of tokens. A simple analogy would be doing a single for loop (hence a single pass of the SIMPLE source) and doing multiple instructions versus multiple for loops

(hence multiple passes of the SIMPLE source) and doing a single instruction within each loop.

Making the codebase much easier to understand and follow, without polluting the parsing logic with the extraction logic is a larger priority. This makes it easier to rectify bugs since the parsing and extraction logic are clearly delineated, and the component responsible for that bug can be easily found.

Design consideration 2: modifying SIMPLE grammar vs using Shunting Yard algorithm to parse expressions

Problem Statement: SIMPLE uses left-recursive grammar in expressions which a normal top-down recursive descent parsing algorithm will not be able to successfully parse. Thus, either the left recursive grammar has to be modified in order for recursive descent to parse it, or a new parsing algorithm must be used.

Constraints of the Problem:

A valid SIMPLE expression must be parsed successfully
The valid AST of the expression should be the output of the successful parse.

Evaluation Criteria:

1. Developer knowledge
2. Code readability

Description of Possible Solutions:

Translate SIMPLE grammar to allow recursive descent to parse expressions
The left recursive grammar can be eliminated by translating it to an equivalent form.

Use Dijsktra's Shunting Yard algorithm
This powerful algorithm can parse any mathematical expression to output the Reverse Polish Notation (RPN) form, which can then be used to build up the AST.

Evaluation of Solutions

Developer knowledge	
Translate grammar	Requires the developers to have a deeper understanding of parsing algorithms and the technical expertise in translating the grammar successfully
Shunting Yard algorithm	Requires less technical expertise since the algorithm can just be followed.

Code readability	
Translate grammar	The recursive descent algorithm does not need to change and can stay as is, making it easier to follow and read.
Shunting yard algorithm	Pollutes the recursive descent algorithm with another unrelated algorithm, potentially causing confusion for other developers. Also requires more checks to be added since Shunting Yard cannot detect some invalid expressions (as discussed in section 3.2.3.2).

Final Choice

Even though translating the grammar may lead to more readable code, the Shunting Yard algorithm was chosen. Since both the query and source parser needs to be able to

parse expressions, both should call the same function to parse the expression. Thus, there is a need to abstract this code out to another file which can be included in both parsers. This effectively eliminates the issue with less readable code while doing the recursive descent algorithm since parsing an expression is just a call to a method that is defined in another file.

Moreover, since the Shunting Yard algorithm produces an RPN, whenever the QE is evaluating the pattern clause, instead of checking if an expression is a valid subtree of another expression, it can instead be checked if an RPN is a sublist of another RPN. The latter is much easier to implement and much less prone to bugs.

3.3 Design Extractor

3.3.1 Overview of Design Extractor

After the program parser has successfully parsed and verified the source program, it creates an AST to be passed to the design extractor. The entry point to the Design Extractor is defined by a single method `ExtractDesigns`. The Design Extractor assumes all logical and semantic checks to be completed in the SIMPLE Parser and thus does not perform any additional checks related to the source code. Therefore the design extractor maintains a single purpose, which is to extract all possible design entities and abstractions from the AST provided. The overall structure is seen below in Figure 3.3.2.1.

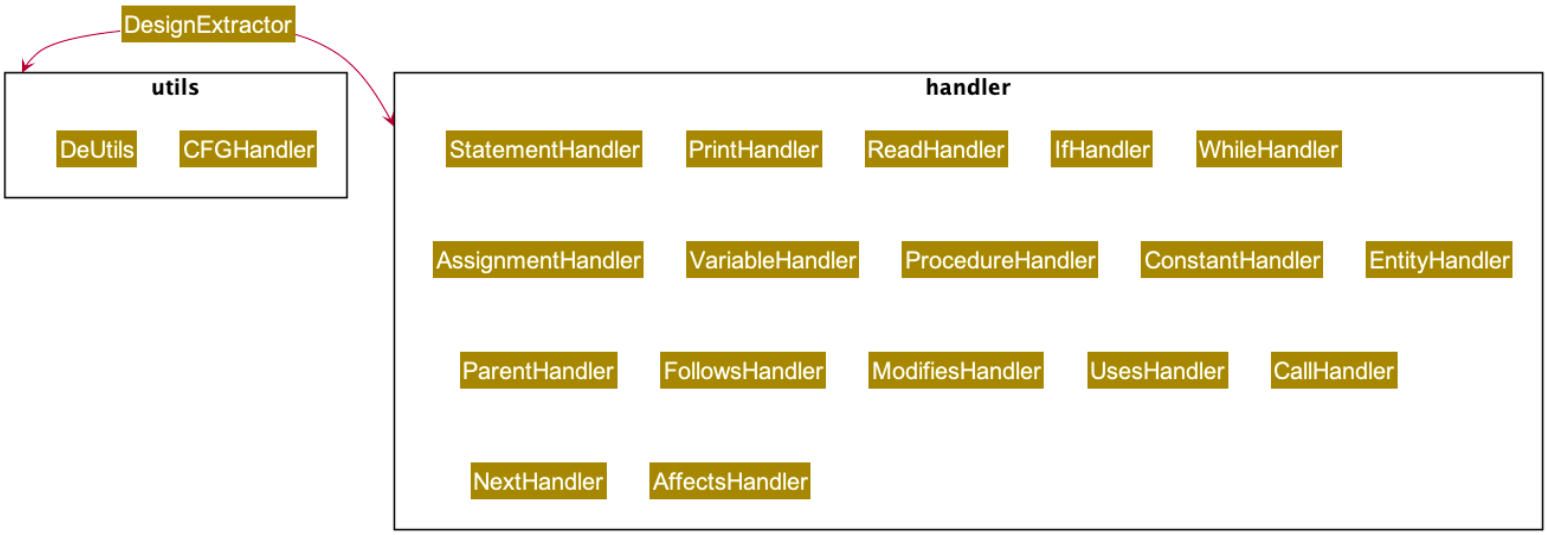

Figure 3.3.2.1 Design Extractor Architecture Diagram

3.3.2 Design of Design Extractor

Structurally, the design extractor consists of multiple handler classes that extract out the relevant information as seen in Figure 3.3.2.1. The individual handlers are generally designed to be independent of each other. However, certain relationships are reliant on another relationship being populated first. The Design Extractor makes use of the chain of responsibility design pattern, which lets us pass requests along a chain of handlers.

Upon receiving a request, each handler decides either to process the request or to pass it on to the next handler in the chain. The dependencies between relationships that rely on another will be further explained in the following sections.

The design extractor utilises a general `BreadthFirstTraversal` function to extract and populate the PKB with design entities and relationships that can be gleaned from a single pass over the AST. More complex entities or relationships which require more than one pass through the AST, or are dependent on other relationships will be extracted independently of the general `BreadthFirstTraversal`.

Extraction of Information

All elements of extraction logic are abstracted out into their respective handlers, where each handler class has a name that is self-explanatory and describes the entity/relationship that it will extract. For example, ReadHandler will extract out all read statements and populate the PKB with the statement number and the variable being read.

The various design entities each have their own handler, namely the StatementHandler, PrintHandler, ReadHandler, IfHandler, WhileHandler, AssignmentHandler, CallHandler, VariableHandler, ProcedureHandler and ConstantHandler. EntityHandler is an additional handler to extract information for the PKB to keep track of what the design entity type is for a particular statement. Additionally, handlers exist for the various design abstractions, namely the ParentHandler, ModifiesHandler, FollowsHandler, CallHandler, UsesHandler, NextHandler and AffectsHandler.

Only the DesignExtractor class is aware of the various handlers, however the handlers themselves are not aware of each other. Therefore the extraction logic is encapsulated within the handler classes themselves, while the DesignExtractor class acts as a facade and calls each of the handlers to extract and populate the PKB. This provides a centralised structure which easily scales as a new handler can be created whenever a new design entity/relationship has to be extracted.

Relation Extraction Order

Certain entity information and relationships, such as If-statements and Follows relation, can be extracted directly from the AST. However, some relations use other relations in their extraction. One example is UsesS/UsesP. The Uses relation is transitive through container statements and function calls. Since it requires Parent/Parent* and Call/Call* information to be extracted, to reduce repeated computation, it should only be called after ParentHandler and CallHandler.

Fig. 3.3.2.1 shows the dependencies among all current handlers at the time of Iteration 3. Do note that the diagram is **not** a UML diagram. An arrow extends from Handler A to Handler B if Handler A has to populate its relations before Handler B. Handlers which are not involved in any dependencies are omitted for brevity.

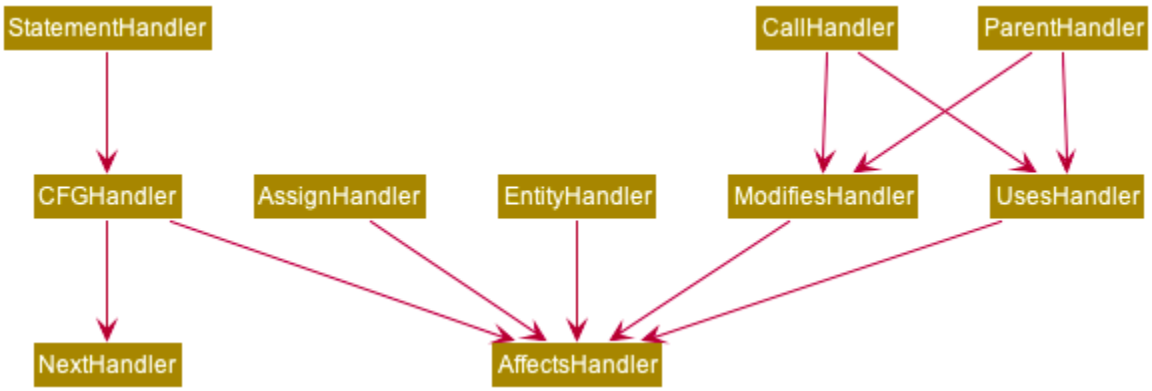

Figure 3.3.2.1 Dependency graph among handlers

The dependency graph among handlers forms a Directed Acyclic Graph (DAG), and thus a topological ordering of the various handlers exists. To ensure that the relations needed by every handler have been populated when it is called, the DE calls the handlers in the topological order.

The current handlers are called in this order:

1. StatementHandler

2. PrintHandler
3. ReadHandler
4. IfHandler
5. WhileHandler
6. VariableHandler
7. ProcedureHandler
8. EntityHandler
9. ConstantHandler
10. AssignmentHandler
11. FollowsHandler
12. ParentHandler
13. CallHandler
14. UsesHandler
15. ModifiesHandler
16. CFGHandler
17. NextHandle
18. AffectsHandler

Population of PKB

Assuming once again the following SIMPLE source code:

```
procedure main {
  read x;
}
```

The following Figure 3.3.2.2 shows the interaction between the DesignExtractor and the PKB while populating the PKB with information about read statements. After the AST is produced by the source parser, the root TNode of this AST is passed to the DesignExtractor via the `ExtractDesigns` method. The `ExtractDesigns` then calls for a `BreadthFirstTraversal` of the AST, activating the respective handlers to populate the PKB.

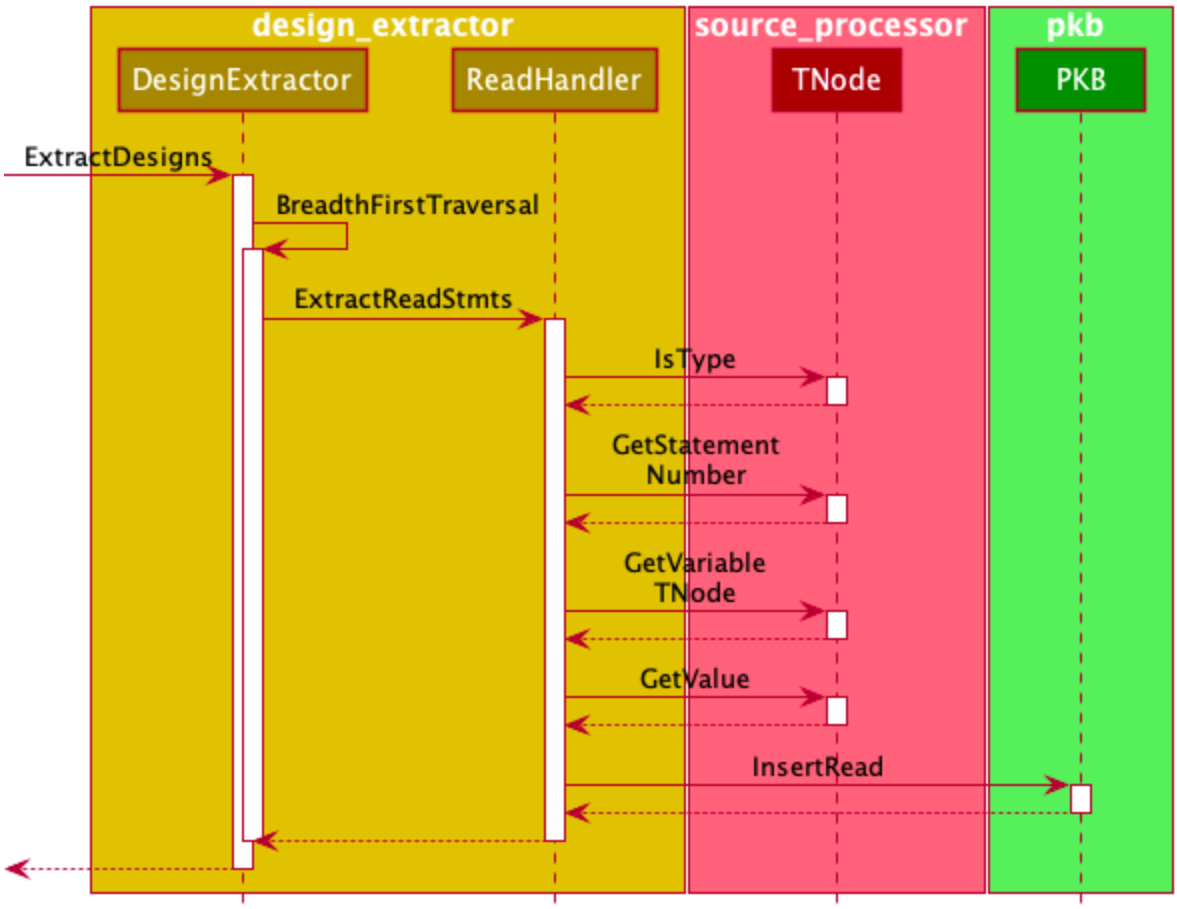

Figure 3.3.2.2 Design Extractor to PKB Sequence Diagram

The `BreadthFirstTraversal` method makes use of the breadth first search algorithm to traverse and visit all TNodes in the AST. For each TNode met within the AST, all handlers are called, but irrelevant handlers will exit the function call and the execution path returns back to the main BFS once the type check fails. It is important to note that no type checks are done within the `BreadthFirstTraversal` directly as the DesignExtractor should not be concerned with the individual extraction logic. Therefore, type checks are done only within their respective handlers.

Within the `BreadthFirstTraversal`, ReadHandler's `ExtractReadStmts` is called, as seen from the diagram above. The type of TNode is then checked using the `IsType` function provided by the TNode class. In this case, the ReadHandler will only proceed if

the TNodeType of this TNode is of the Read type. In the event that the `IsType` function returns false, the `ExtractReadStmts` will exit back to the main BFS and the next handler will be called. The type checking for the irrelevant handlers are not shown here in this sequence diagram as it will render the diagram unreadable, but within the code implementation, all handlers will be called and the type check will be done internally.

Once the TNodeType has been verified to be of Read type, the `ExtractReadStmts` function will then extract the statement number of the TNode from the AST using `GetStatementNumber`. To get the value of the variable being read into, i.e. the `a` in `read a`, the Variable TNode, which is the child of the Read TNode, must first be retrieved. This is retrieved via `GetVariableTNode`. Finally, the value of the Variable TNode is extracted using `GetValue` and together with the statement number, this information gets populated into the PKB's ReadTable using `InsertRead`.

3.3.3 Design Decisions for Design Extractor

Design Consideration 1: Multiple BFS vs General BFS

Problem Statement: Choice between having a general Breadth First Search (BFS) that traverses the AST once versus having multiple BFS that traverse the AST once within each BFS

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Design entities and abstractions must be extracted
Design entities such as: Constants, variables, statement numbers, procedures Read statements, print statements, if statements, while statements, assignment statements, call statements

Design abstractions such as: Follows/Follows* relationships, Parent/Parent* relationships, Uses relationships, Modifies relationships, Calls/Calls* relationships, Next/Next* relationships, Affects/Affects* relationships

Evaluation Criteria:

1. Ease of adding future extensions
2. Performance
3. Adherence to design principles

Description of Possible Solutions:

Multiple BFS methods that are independent of each other
Each function traversing the AST using breadth first traversal to extract and handle the design entities and relationships
General BFS that calls respective handlers
Traverses the AST once, calls the respective handlers to extract and handle the design entities and relationships

Evaluation of Solutions

Ease of adding future extensions	
Multiple BFS	Easier to develop initially, just have to insert a brand new BFS method whenever a new entity/relation has to be extracted. However, this added to the clutter in the main DesignExtractor file, making the addition of future extensions difficult as readability would be compromised with so many similar methods in the main file.

General BFS	Harder to set up initially, as handler methods have to be abstracted out as individual methods. However, it is easy to extend as a new handler would be added every time a new entity/relationship has to be extracted. Not only so, the readability would be greatly improved as there are less code overlaps, adding to the ease of creating new extensions.

Performance	
Multiple BFS	Traverses the AST multiple times using the BFS algorithm for every relation. Results in unnecessary computation cycles for the program to keep traversing down the AST. Not the most sustainable as the amount of requirements grows.
General BFS	Traverse the AST once, saving time and is able to extract multiple relations at once by calling their respective handlers.

Adherence to design principles	
Multiple BFS	Each handler has to make sure that the BFS algorithm within is correct, in addition to the responsibility of extracting the design entities/relations. This violates the Single Responsibility Principle (SRP) which states that the handler should only care about the handling of their respective designs. Ensuring that the BFS algorithm is correct is therefore a violation of SRP.
General BFS	The general BFS method handles the BFS algorithm, while the handles do not have to care about the traversal of the AST, respecting that their sole responsibility is to extract the respective design entities/abstractions.

Final Choice

Having a general BFS was our final choice, due to the fact that it would be easier to extend once the number of requirements grows, as well as creating a more readable code base for us. Additionally, the performance would be improved with less time being spent on traversing the AST again and again, needlessly. Finally, it adheres to the single responsibility principle as the extraction handlers are freed of the responsibility of the traversal logic.

Design Consideration 2: Handler Methods vs Handler Classes

Problem Statement: Choice of structural organisation of handler code

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Handler implementation must be encapsulated
Regardless of the structural organisation of the handler classes/methods, the handler implementation must be restricted and encapsulated within their own class/method
Design entities and abstractions must be extracted
Similar to Design Decision 1, the design entities and abstractions must still be extracted correctly and populate the PKB as such

Evaluation Criteria:

1. Ease of adding future extensions
2. Maintainability

3. Adherence to design pattern

Description of Possible Solutions:

Handler Methods
Design extraction logic is abstracted into handler methods within the main DesignExtractor.cpp. Every method is kept within the main file.
Handler Classes
Instead of calling handler methods in the immediate DesignExtractor scope, handler classes would have to be used, and the methods within that class will have to be called in order to extract designs.

Evaluation of Solutions

Ease of adding future extensions	
Handler Methods	Easier to implement initially. A new method can be added to the main DesignExtractor file every time an extension/feature is added
Handler Classes	Also easy to extend as a new class can be created whenever an extension is required. However the filesystem will appear more cluttered as files have to be added whenever a new class is created. This includes the header (.h) file and the actual .cpp file

Maintainability	
Handler Methods	Bad for maintainability. File can grow to be too long and unreadable, especially as more complicated relationships like Use and Modifies are extracted

Handler Classes	Even though more files are created, these files are smaller and therefore easier to maintain and debug. The logic would also be clearly encapsulated within their own classes and decoupled from the logic of other handlers

Adherence to design pattern	
Handler Methods	All handler logic is exposed to the Design Extractor class, no facade pattern is being used
Handler Classes	All handler logic is hidden from the Design Extractor class, the handler classes act as a facade for the Design Extractor to make use of it.

Final Choice

Having multiple handler classes instead of methods was our final choice. Accounting for the fact that many more relations have to be extracted (and with some of them highly non-trivial), it would be better in the long run to have abstracted out the handlers in different classes rather than having one file to handle all extraction logic. It would also be easier to maintain the smaller code files for the handler classes rather than maintain a larger main Design Extractor class.

3.4 PKB

3.4.1 Overview of PKB

The PKB is a centralised storage component which contains all design entity and design abstraction tables. It also provides key table operations via concrete APIs provided in `PKB.h` so that the Design Extractor and Query Evaluator can populate or extract information easily. When the Design Extractor or Query Evaluator calls a PKB's API, the respective entity or abstraction table involved in that API call will invoke its own API to execute the operation under the hood. The design of the PKB will be further elaborated in the following section.

3.4.2 Design of PKB

The PKB is a class representation of the Program Knowledge Base, a body which stores all design entities and their relationships to supply to the query on demand. The PKB is thus designed as a container of various design entity tables as well as relationship tables. When the PKB class is instantiated, all the underlying tables will be initialized to be empty as well. Figure 3.4.2.1 provides an overall view of the current design entity and abstraction tables stored in the PKB.

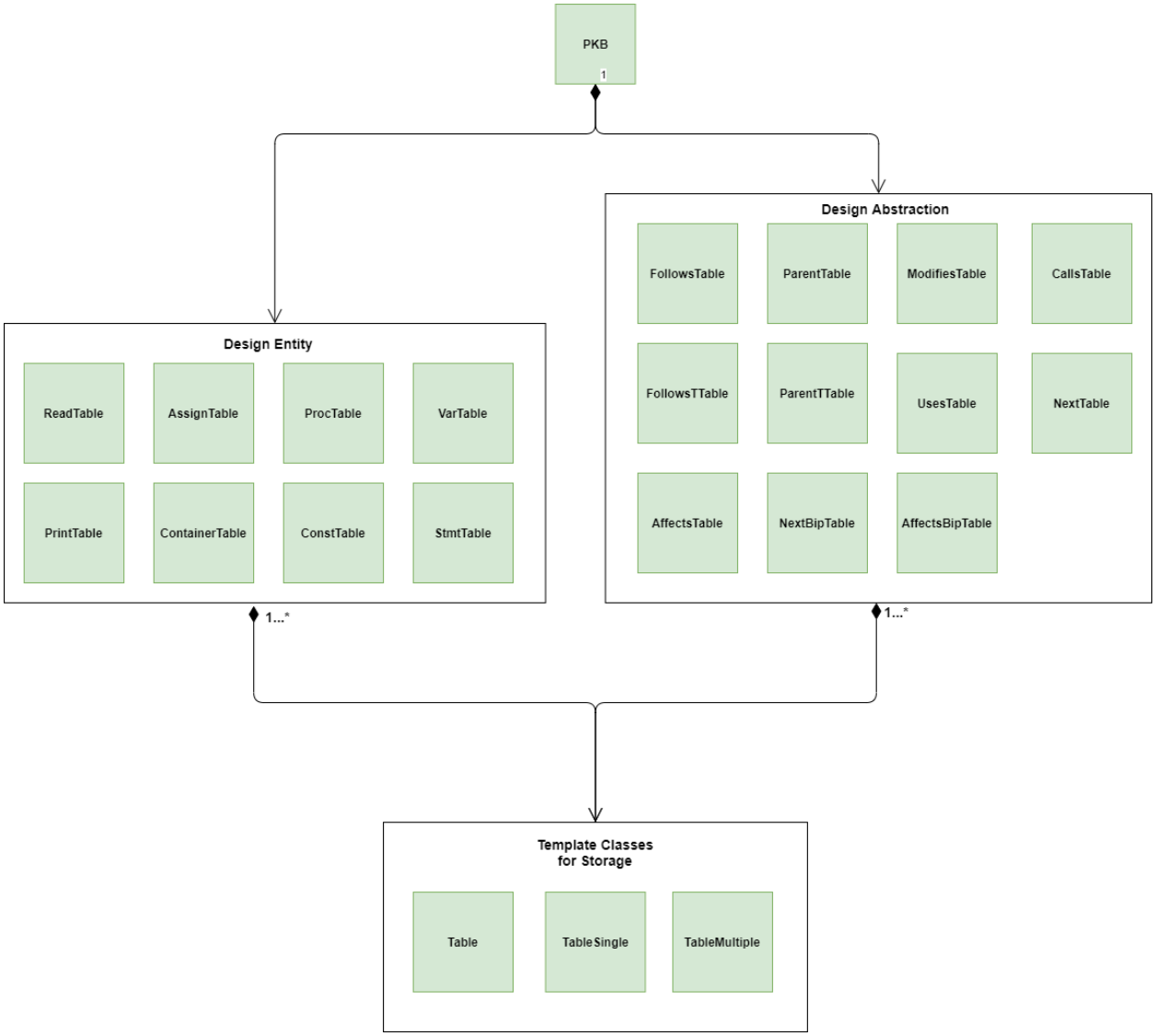

Figure 3.4.2.1 Architectural diagram of PKB

For design entity storage, the PKB consists of `VarTable`, `StmtTable`, `ConstTable`, `ReadTable`, `PrintTable`, `ContainerTable`, `ProcTable` and `AssignTable`. For design abstraction storage, the PKB contains `FollowsTable`, `FollowsTTable`, `ParentTable`, `ParentTTable`, `ModifiesTable`, `UsesTable`, `CallsTable`, `CallsTTable`, `NextTable`, `AffectsTable`, `NextBipTable`, `AffectsBipTable`. Each table contains their own sets of APIs and as such can be invoked by the PKB when needed. A PKB facade also combines all relevant APIs belonging to each table

into separate sections with extensive comments and documentation to support APIs reference and usage from other components.

3.4.3 Table Storage and Retrieval

3.4.3.1 Usage of Template Class

Being a storage component, the PKB supports all common table operations which include *Insertion*, *Retrieval* and *Clear*. However, different design entities or relationships require different parameters to be inserted, different forms of retrieval (such as retrieve all or retrieve with a parameter) as well as different operations specific to its definition. For instance, design abstraction tables need to support some additional operations such as relationship checking. This prompted the need to encapsulate each design entity and design abstraction into a separate class. At the same time, different classes may still require similar storage data structures. To ensure flexibility in the design of each entity or abstraction class, three categories of table storage which cover most use cases are derived so that each class can use Composition to mix and match the tables for their needs. These tables are implemented with C++ template classes to accommodate generic data types so that they can be shared.

Table Characteristic	**Template Name**	**Data Structure**
Pure Container	`Table`	`Set<K>`
Key - SingleValue Mapping	`TableSingle`	`Map<K, V>`
Key - MultipleValues Mapping	`TableMultiple`	`Map<K, Set<V>>`

Each category of tables can share similar operations but their implementation and functionalities vary due to the differences in their storage structure. They are implemented using C++ Standard Template Library Associative Containers as the underlying data structures.

`Table` Template Class

For the `Table` template class which is a pure container, `std::unordered_set<K>` is the data structure used. Design entity tables which only need to store their entity values for the entire program such as ConstTable, StmtTable and VarTable utilise this template `Table` for storage. Due to its simplicity, `Table` only supports the following operations:

Storage	Insert value into set
Retrieval	Get all values in set
Clear	Clear all values in set

`TableSingle` Template Class

For the `TableSingle` template class, it is used in situations where each key is only mapped to a single value. `std::unordered_map<K, V>` is the data structure used to support this class. Entity or abstraction tables which use `TableSingle` through Composition include `ReadTable`, `PrintTable`, `ProcTable`, `AssignTable`, `FollowsTable`. To illustrate, the following table shows a representation of the storage in PrintTable and FollowsTable enabled by `TableSingle` template class.

PrintTable		Sample SOURCE Program	FollowsTable	
stmt	**variable**	1. print width; 2. print length; 3. print center;	**stmt1**	**stmt2**
1	'width'		1	2
2	'length'		2	3
3	'center'		3	4

4	‘area’	4. print area;		

In the above example, the `PrintTable` uses `TableSingle` template class via Composition to store a mapping of `<stmt, variable>` since each print statement can only contain one single variable. Using the `TableSingle` template, the `PrintTable` has essentially stored the following information for queries related to print statements:

1. Whether a statement index is a print statement
2. The variable used in each print statement
3. All print statements present in the program
4. All variables printed in the program

The `TableSingle` template table is thus designed to provide all generic operations which can contribute towards answering the above possible queries. The following table describes generic operations from `TableSingle` and which query number they can possibly support.

Operation	Description	Possible Support
Storage	Insert a `<key, value>` mapping	
	Check if a key is present	[1]
Retrieval	Get the corresponding value given a key	[2]
	Get all keys	[3]
Clear	Clear all `<key, value>` mappings	

For more specific queries, the entity or abstraction classes such as `PrintTable` can use existing `TableSingle` operations to first retrieve and then manipulate the result further.

`TableMultiple` Template Class

For the `TableMultiple` template class, the intention and implementation is almost similar to `TableSingle`, except that each key is mapped to a set of values instead. This is to support design entities for which the same relationship can hold between itself and multiple instances of another entity. For instance, a `stmt` entity can modify multiple `variable` entities within that same statement. This can be captured in the `ModifiesTable` as a mapping of `<stmt_index, set<variable>>`. The `TableMultiple` thus provides operations to support such relationships. Entity or abstraction tables which use `TableMultiple` through Composition include `CallsTable`, `FollowsTTable`, `ParentTable`, `ParentTTable`, `UsesTable`, `ModifiesTable`, `ContainerTable (while/if)`, `NextTable`, `AffectsTable`, `NextBipTable`, `AffectsBipTable`.

To illustrate, the following table shows a representation of the storage in `ModifiesTable` and `UsesTable` enabled by `TableMultiple` template class.

ModifiesTable		Sample SOURCE Program	UsesTable	

stmt	**variables**	1. x = 2;
		2. y = x;
		3. z = x + y
		4. if (x > 1) then {
		5. a = x + 1; }
		else {
		6. b = x + 2; }
			stmt	**variables**

1	{‘x’}		1	{}
2	{‘y’}		2	{‘x’}
3	{‘z’}		3	{‘x’, ‘y’}
4	{‘a’, ‘b’}		4	{‘x’}
5	{‘a’}		5	{‘x’}

6	{‘b’}		6	{‘x’}

The generic operations provided by `TableMultiple` is as follows:

Operation	Description
Storage	Insert a `<key, value>` mapping and update the value set
	Check if a key is present
Retrieval	Get the corresponding set of values given a key
	Get all keys
Clear	Clear all `<key, Set<value>>` mappings

Overall PKB table structures

The following table goes into details about the data types and data structures used for each PKB table.

Design Entity Tables		
Table	**Storage**	**Structure**
`VarTable`	Container for variables	`Set<string>`
`ConstTable`	Container for constants	`Set<int>`
`StmtTable`	Container for statements	`Set<int>`
`ReadTable`	Map stmt_no to var_name	`Map<int, string>`
`PrintTable`	Map stmt_no to var_name	`Map<int, string>`
`ContainerTable`	Map stmt_no to a set of var_names	`Map<int, Set<string>>`
`EntityTable`	Map stmt_no to entity type	`Map<int, string>`
`AssignTable`	Map stmt_no to token_list	`Map<int, token_list>`

	Map stmt_no to modified var_name	Map<int, string>
ProcTable	Map proc_name to <start_idx, end_idx>	Map<int, pair<int, int>>
Design Abstraction Tables		
Table	**Storage**	**Structure**
FollowsTable	Map stmt_no to stmt_no	Map<int, int>
FollowsTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
ParentTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
ParentTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
ModifiesTable	Map stmt_no to a set of var_names	Map<int, Set<string>>
	Map proc_name to a set of var_names	Map<string, Set<string>>
UsesTable	Map stmt_no to a set of var_names	Map<int, Set<string>>
	Map proc_name to a set of var_names	Map<string, Set<string>>
CallsTable	Map stmt_no to var_name	Map<int, Set<string>>
	Map proc_name to a set of proc_names	Map<string, Set<string>>
CallsTTable	Map proc_name to a set of proc_names	Map<string, Set<string>>
NextTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
NextTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
AffectsTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
AffectsTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
NextBipTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
NextBipTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
AffectsBipTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>
AffectsBipTTable	Map stmt_no to a set of stmt_nos	Map<int, Set<int>>

3.4.3.2 Execution of PKB's APIs

Overall, the PKB is a class which references one object of each design entity and abstraction class via Composition. Its APIs will invoke the corresponding entity or

abstraction class' APIs. Each entity or abstraction class in turn references one or multiple template table objects as the underlying storage structure and their APIs are built on top of the generic operations from the template tables. To use the PKB's APIs, the PKB is first instantiated as an object in TestWrapper.h. This in turn instantiates all the table classes. This PKB instance then becomes the entry point for other components such as Design Extractor or Query Evaluator to call its concrete APIs for insertion, retrieval or relationship checking operations. The following figures 3.4.3.1 and 3.4.3.2 are sequence diagrams which showcase the interaction of PKB with the Design Extractor and Query Evaluator.

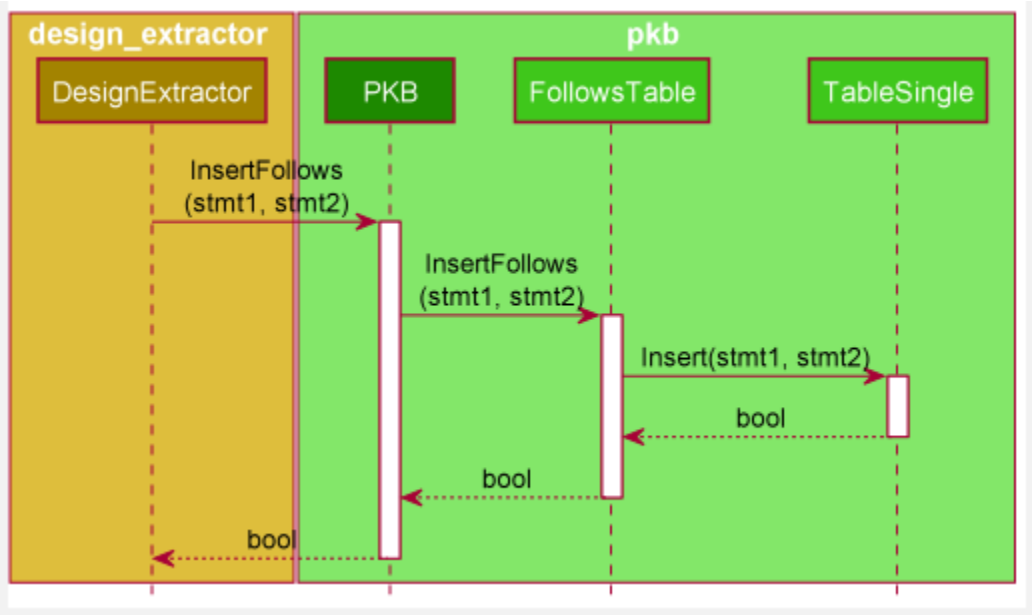

Figure 3.4.3.1 Sequence diagram of overall execution of PKB with Design Extractor

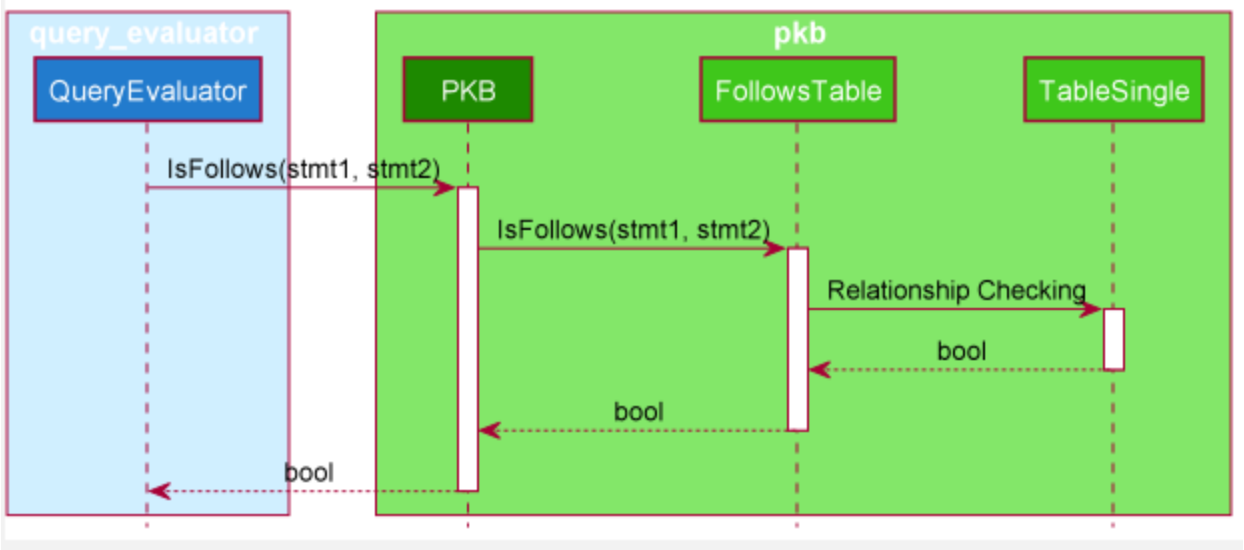

Figure 3.4.3.2 Sequence diagram of overall execution of PKB with Query Evaluator

3.4.4 Design Decisions for PKB

Problem Statement: Choice of data structure for storage

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Represent the relationship between entities
The data structure must accommodate and represent different possible relationships that exist between different entity types.
Capture sufficient information needed to answer queries
The data structure must not miss out any important information from each design entity and abstraction necessary for query answering.
Efficient insertion and retrieval

The data structure should support efficient insertion and retrieval of different entities

Evaluation Criteria:

1. Information related to the relationships between entities are stored
2. Population from Design Extractor and extraction from Query Evaluator fully supported
3. Time complexity of insertion and retrieval

Description of Possible Solutions

With the use of `pair` and `list`
Design entities and abstractions can be stored using other data structures. For design entity tables which only require storage of all values of entity type `ent`, `std::vector<ent>` can be simply used. For a table which contains the relationship between two entity types `ent1` and `ent2` such as `FollowsTable`, a list of pairs such as `std::vector<std::pair<ent1, ent2>>` can be used for storage. For tables which contain relationships between an entity `ent1` and multiple instances of another entity `ent2` such as Calls, a data structure such as `std::vector<std::pair<ent1, std::vector<ent2>>>` can be used instead. Each pair is an instance of entity type `ent1` and a list of instances of entity type `ent2`.
With the use of STL Associative Containers
For the three storage requirements mentioned above, `std::set<ent>`, `std::map<ent1, ent2>` and `std::map<ent1, std::set<ent2>>` are used.
With the use of STL Unordered Associative Containers
For the three storage requirements mentioned above, `std::unordered_set<ent>`, `std::unordered_map<ent1, ent2>` and `std::unordered_map<ent1,`

`std::unordered_set<ent2>>` are used.

Evaluation of Solution

Represent the relationship between entities	
With the use of `pair` and `list`	This implementation can capture the relationship between entities. For instance, a `std::pair<stmt1, stmt2>` in `FollowsTable` suggests a relationship `Follows<stmt1, stmt2>`. Meanwhile, a `std::pair<stmt1, std::vector<var_name>>` in `ModifiesTable` suggests all `Modifies` relationships that `stmt1` has with instances of `variable`.
With the use of STL Associative Containers	`std::map<ent1, ent2>` and `std::map<ent1, std::set<ent2>>` can store and represent the relationship between entities in a similar way.
With the use of STL Unordered Associative Containers	`std::unordered_map<ent1, ent2>` and `std::unordered_map<ent1, std::unordered_set<ent2>>` can store and represent the relationship between entities in a similar way. The only difference is that the order is not preserved in both `std::unordered_map and std::unordered_set.`

Capture sufficient information needed to answer queries	
With the use of `pair` and `list`	Efficiency aside, this data structure allows for extraction of all information stored. For the most complex case which is to extract information from a vector of pairs of `<entity, vector>` which is `std::vector<std::pair<stmt1,`

	`std::vector<var_name>>>`, it is still possible to so by iterating through all pairs within the list and in the worst case continuing to iterate through all entities within each vector stored in each pair.
With the use of STL Associative Containers	Similarly, the total amount of data stored is the same between this implementation and the other two.
With the use of STL Unordered Associative Containers	It is also the same case for this data structure.

Efficient insertion and retrieval	
With the use of `pair` and `list`	This method is the slowest in terms of insertion and retrieval. This is because it requires linear time `O(N)` to iterate through the list of pairs to search for the desired pair.
With the use of STL Associative Containers	The data structures used in this method provide a significant improvement in terms of time complexity as compared to the above implementation. This is because the `std::set` and `std::map` in C++ STL are implemented using self-balancing BST. As such, the search time for each table mapping will have time complexity of `O(logN)` as the tree is height-balanced and so runtime is proportional to the height of the tree. At the same time, insertion operations will also require `O(logN)` plus some costs related to rebalancing of the tree.
With the use of STL Unordered	The data structures used for this method is on average the most efficient in terms of time complexity. This is because

Associative Containers	`std::unordered_set` and `std::unordered_map` are implemented by a hash table and thus assuming that there is a sufficient number of buckets as well as the universal hashing assumption holds true, the amortized cost for insertion and search time will be `O(1)` on average.

Final Choice: The final choice is to use STL Unordered Associative Containers. It has been shown that generally in terms of information storage all three design choices are similar to one another. Thus, time complexity analysis is used to determine the most efficient data structures. For the final choice with Unordered Associative Containers, another rationale besides them having better time complexity than other options is that the order of query output results does not matter. As such, the Query Evaluator does not require the PKB to maintain the order of data upon sequential scanning of the SIMPLE source program and population from the Design Extractor. This leaves the PKB to freely utilize the most efficient data structures for storage and retrieval. As the number of entities and relationships to be stored scale with longer SIMPLE source programs across iterations, the benefits of having better time complexity on average for key table operations will also scale significantly to meet the requirements of each iteration.

Problem Statement: Retrieval of inverse relationship

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

A Query clause may provide either entity and request for the other
A query SuchThat or Pattern clause may provide either the left or right entity and the PKB tables need to support efficient retrieval of the remaining entity.

Each entity or abstraction table needs to support `GetAll` operations
For instance, an abstraction table such as `CallsTable` needs to support both operations `GetAllProceduresThatCalls` to get all procedures which calls one or more other procedures and `GetAllCalledProcedures` to get all procedures which are called by one or more other procedures.

Evaluation Criteria:

1. Time complexity of retrieval
2. Space complexity to support the design choice

Description of Possible Solutions

No additional template tables used within each entity or abstraction table
In this design pattern, the existing template tables such as `Table`, `TableSingle`, `TableMultiple` currently contained in each entity or abstraction table such as `ModifiesTable` or `UsesTable` which stores a direct relationship will also be used to extract the inverse relationship. For instance, a statement `stmt_index` is currently mapped to a set of variables `{v1, v2, v3}` in the `UsesTable` as it modifies all those variables. A query such as `GetAllVarsUsed(stmt_index)` can be directly extracted by returning the set of variables stored in the bucket where `stmt_index` is hashed to. However, for the inverse query `GetAllStmtsThatUses(v1)`, all statement indexes in the hash table needs to be scanned and for each index, all values mapped to it will be scanned through to search for `v1`. All statement indexes which `Modifies v1` will be returned.
Include additional template tables for inverse relationship storage
In this implementation, for each `TableSingle` and `TableMultiple` for relationship storage in a design entity or abstraction class, there will be an additional template

table to store the inverse relationship. For example, the `FollowsTable` will compose two `TableSingle` objects instead of one: a `follows_table` and `inverse_follows_table` objects. Each time a relationship `Follows(stmt1, stmt2)` is inserted, a mapping of `<stmt1, stmt2>` will go to the `follows_table` and the inverse mapping of `<stmt2, stmt1>` will go to the `inverse_follows_table`.

Evaluation of Solution

Time complexity of retrieval	
No additional template tables used within each entity or abstraction table	This implementation is very inefficient for retrieval of inverse relationship as well as getting all the values as the time complexity for both operations will be `O(N*M)` assuming there are N rows and M columns.
Include additional template tables for inverse relationship storage	For this implementation, `GetAllStmtsThatUses(v1)` will only be `O(1)` since `v1` is now a key of the inverse table and can be hashed to find the bucket which contains all statements that `Uses v1`. For the `CallsTable` example, `GetAllCalledProcedures` will also be reduced to `O(N)` instead of `O(N*M)` because it can be achieved using the `GetAllKeys` operation provided by the inverse operation.

Space complexity to support the design choice	
No additional template tables used within each	This implementation requires no additional space for storage as only the direct relationships are stored.

entity or abstraction table	
Include additional template tables for inverse relationship storage	This implementation will require in the worst case one additional table for each direct relationship with the same table size. As such, the space complexity doubles.

Final Choice: Additional template tables are chosen because advanced SPA requirements allow pre-population of the PKB and do not have a hard limit on the space complexity. As such, it is more reasonable to trade-off double space complexity for much better runtime which can support the Query Evaluator a lot more efficiently. Besides, it is also made known that the SIMPLE source program will be about 500 lines which is still reasonable for the PKB to store additional tables.

3.5 Query Processor

3.5.1 Overview of the Query Processor

The Query Processor is split up into three main components: the Query Parser, Query Evaluator and Query Projector. The entry point to the Query Processor is done via a single API: **`ProcessQuery(query_string, PKB)`**, which supplies the Query Processor with the desired query in string format, as well as the PKB, pre-populated by the Source Processor.

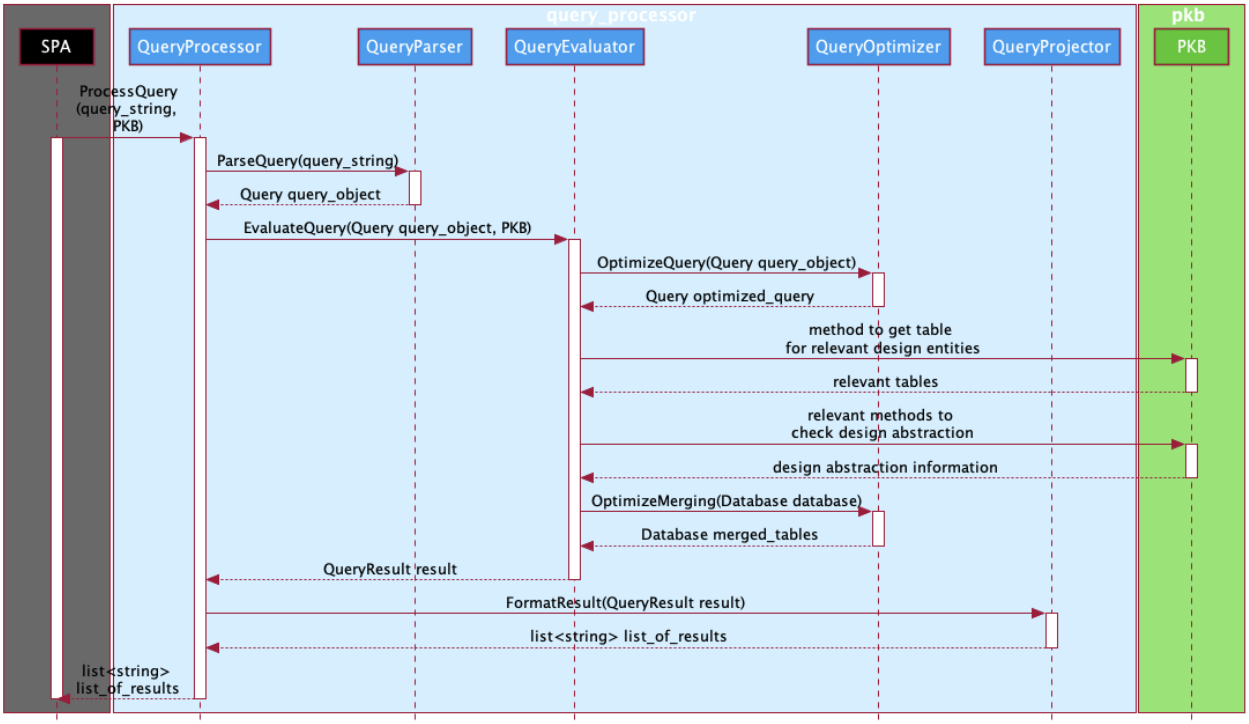

Figure 3.5.1.1 Sequence diagram of overall execution of Query Processor

The Query Processor then sends the query string to the Query Parser via a `GenerateQuery(query_string)` call, which returns a Query object to the Query Processor. Information about the Query class can be found in Section 3.5.2, and further details on the implementation of the Query Parser will be covered in Section 3.5.3.

The Query object is then evaluated in the Query Evaluator, which also needs to be supplied with the pre-populated PKB in the `EvaluateQuery(query_object, PKB)` API.

The Query Evaluator passes the Query object on to the Query Optimizer to be sorted and preprocessed using `OptimizeQuery(query_object)`.

The Query Evaluator then calls the relevant APIs to retrieve information on the selected design entities. Implementation details on the Query Evaluator will be covered in Section 3.5.4.

The Query Evaluator then passes the Database, which is a list of tables representing the intermediate result space to the Query Optimizer via `OptimizeMerging(Database tables)`, which merges the tables, producing the final result of the query to the Query Evaluator.

Finally, the Query Evaluator returns the result of the query in a QueryResult object to the Query Projector, which simply formats the results into a list of strings as required by the Static Program Analyzer.

3.5.2 Query Class

3.5.2.1 Overview of the Query Class

The Query class is an object representation of the query string, and captures all necessary information to evaluate a query. It is the only shared data structure between Query Parser and Query Evaluator. The two main components of the Query class are:

1. A list of SelectedEntity objects
2. A list of Clause objects

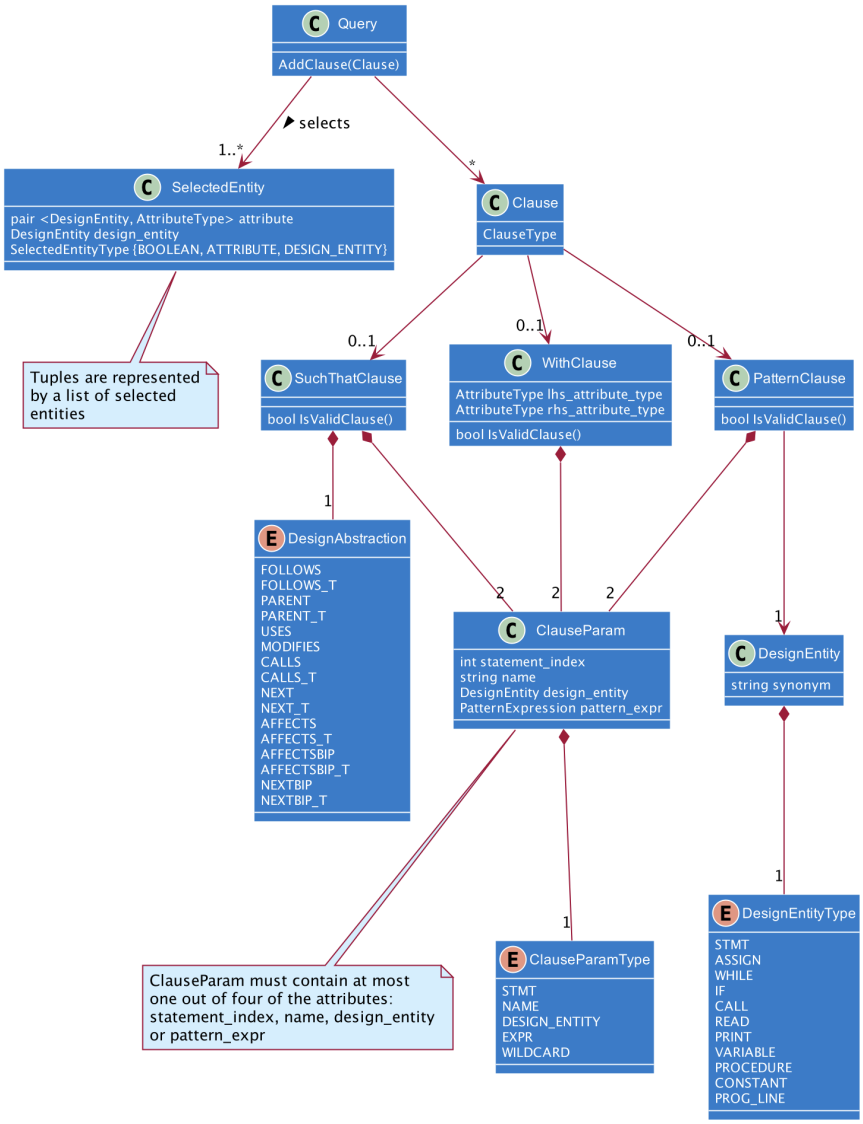

Figure 3.5.2.1 Class diagram of the Query class

3.5.2.2 Data Structures in the Query object

Selected Entity

The selected entity encapsulates information about what the query is seeking to select. The table below summarizes how each type of selection in PQL is represented by the SelectedEntity object.

Type	**Data Representation**
BOOLEAN	SelectedEntityType::BOOLEAN
Synonyms	SelectedEntityType::DESIGN_ENTITY, and a DesignEntity object which contains the synonym string being selected as well as the enum DesignEntityType. (e.g `assign a; Select a` would be represented by a DesignEntity object containing DesignEntityType::ASSIGN and synonym string “a”)
Attributes	SelectedEntityType::ATTRIBUTE, and a pair containing a DesignEntity object and an AttributeType enum representing the attribute type. e.g `print pn; Select p.varName` would be represented by: `pair<DesignEntity(DesignEntityType::PRINT, “pn”), AttributeType::VAR_NAME>`
Tuple	Instead of being represented by a single SelectedEntity object, a tuple is a list of SelectedEntity objects in the Query class, where the SelectedEntity could be of type ATTRIBUTE or DESIGN_ENTITY. If the query object contains more than one SelectedEntity object, the selection of a tuple can be identified.

Clause

The Clause object is a wrapper that contains only one of either of the three clauses: SuchThatClause, PatternClause and WithClause, and has an associated enum ClauseType which indicates the type of clause the Clause object contains. The query object contains a list of Clause objects to be evaluated.

All Clause objects contain two ClauseParams, one for the left hand side parameter and one for the right hand side parameter (`e.g with LHS = RHS or pattern a(LHS, RHS)`). The ClauseParam struct was designed to be a wrapper around the information represented in clause parameters. Each ClauseParam can only contain one type of attribute (int, string, DesignEntity or PatternExpression). This allows for the Query Evaluator to easily extract information on the type of parameters being evaluated and handle them differently (e.g a statement index “1” might be evaluated differently from a statement synonym “s”).

It was noted that pattern if clauses in fact contain three parameters. However, since two of the three parameters are always designed to be wildcard parameters, it was unnecessary to add a third ClauseParam to the pattern clause unless an extension of the pattern if clause is designed to allow for other types of parameters.

Each clause object contains some extra information specific to the type of clause:

SuchThatClause	DesignAbstraction (e.g FOLLOWS_T, MODIFIES etc.) to identify the relationship being evaluated
PatternClause	DesignEntity to capture information about different design entity types (e.g pattern assign/while/ifs) as well as different synonyms for the same design entity (e.g `assign a, a1; pattern a(v, _)` might have a different result space from `pattern a1(v, _)`)
WithClause	AttributeType (VAR_NAME, PROC_NAME, STMT_NO, VALUE,

	NAME, INTEGER). Each WithClause contains two instances of pair<ClauseParam, AttributeType>, one for LHS and one for RHS.

3.5.2.3 Design Decisions for the Query Class

Problem Statement: Choice of data classes to represent clauses

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Distinguishable clauses
The Query Evaluator must be able to easily distinguish the three different types of clauses (SuchThat, Pattern and With)
Encapsulates all information necessary to evaluate the clause
Primarily, the clauses must include the parameters, and also the different information specific to each type of clause (e.g attribute for with clauses)

Evaluation Criteria:

1. Abstraction of information and minimize unnecessary data in each object
2. Ease of extension to allow QueryOptimizer to compare between different clauses

Description of Possible Solutions

Parent Clause Class
Create a parent clause class that contains the common attributes (e.g ClauseParam) and common methods and have all the different clauses extend the parent clause class

Separate objects for each type of Clause
Create three different objects (SuchThatClause, PatternClause, WithClause) and store them in separate lists in the Query object.
Clause object wrapper
Create a Clause object that stores different types of clauses and a tag to identify the type of clause

Evaluation of Solution

Abstraction of Information, minimize unnecessary information	
Parent Clause class	Good abstraction for common attributes and methods. However, the shared attributes and methods between each type of clause are limited to simply the getters and setters for ClauseParam. In addition, there are a number of methods and attributes that are specific to each type of Clause (e.g DesignAbstraction for SuchThatClause). When the Query Evaluator receives a Clause object, it would have to identify the clause type (using perhaps an enum in the parent Clause object) and typecast the object to the child object before calling these methods.
Separate objects	Minimal unnecessary information in each clause. However, poor abstraction as common attributes are not abstracted out and can be cumbersome to handle as the Query object would have to store the clauses in separate lists.
Clause wrapper	Some unnecessary information included in each wrapper, as there would be uninitialized clauses of other types (e.g a Clause containing a SuchThatClause might have uninitialized

	WithClause and PatternClause).

Allows for comparison between different classes	
Parent Clause class	Easy comparison of shared attributes (i.e ClauseParams), but might be difficult when considering attributes belonging to specific type of clauses (e.g the DesignEntity of the pattern clause) as it would require the comparator to recognise the type of clause and extract the necessary information (type casting might also be involved here)
Separate objects	Difficult comparison between different types of clauses (e.g comparing a SuchThatClause to a PatternClause). It is also difficult to communicate to the Query Evaluator the order with the clauses should be evaluated (e.g intertwined SuchThatClause and PatternClause evaluation) as the clauses are stored in separate lists.
Clause wrapper	Easy comparison of shared attributes (i.e ClauseParams). Easier comparison when considering attributes belonging to specific types of clauses as the Clause already contains the specific PatternClause, SuchThatClause or WithClause object, hence it might be easier to implement a comparator function.

Final Choice: Clause wrapper class. The Clause wrapper class gives the most space and flexibility for extension in view of the QueryOptimizer, at the same time avoiding complex inheritance implementation details such as type casting and usage of parent class pointers.

3.5.3 Query Parser

3.5.3.1 Overview of Query Parser

The Query Parser is responsible for the parsing of input queries. It performs two main roles:

1. Syntactic and semantic validation of the query string.
2. Creation of a Query object that reflects the salient elements of the query string.

The exposed method of the Query Parser is the method `ParseQuery`, which accepts the raw input query string and outputs a Query object. The Query object will then be passed to the Query Evaluator, which will then evaluate the result of the query.

The main algorithms of the Query Parser will be discussed in the following section.

3.5.3.2 Design of Query Parser

To illustrate the entire process of the Query Processor, the same query will be used in the examples of the Query Parser, the Query Evaluator, and the Query Projector. Over the course of the three sections, the raw query string will be parsed, evaluated, and have its results formatted as output.

The raw query string to be used as an example is:

```
assign a1; assign a2; constant c; print pn; variable v;
Select <a1, pn.varName> such that Uses(a1, v) pattern a2(v, _)
                          with c.value = pn.stmt# and 1 = 1
```

Note that the colours used here are **not** related to the colours used in the diagrams of this report. They are only used to highlight the 3 possible sections of each query, for ease of reference. This colour scheme is only relevant for this Query Parser section.

The sections of the query that are represented by colour are:

- The **declaration clauses**, which are coloured red and represent all declarative statements used to assign design entity types to synonym names;

- The **selected entities** (referred to as *Selected Entities* as a proper noun), which are coloured green and represent the type of entity to be retrieved from the PKB;
- The **conditional clauses**, which are coloured blue and represent the ‘such that’, ‘pattern’, and ‘with’ clauses used to establish constraints on the data retrieved.

The parsing of the raw query string is therefore separated into three distinct stages as shown below.

1. Parse and validate the declaration clauses.
2. Parse and validate the selected entities.
3. Parse and validate the conditional clauses.

If at any point, any validation check fails, the entire query is deemed invalid. The failure of a validation check can either be due to a syntactic error or a semantic error. It should be noted that in the case of queries which select BOOLEAN, a FALSE result is expected if a semantic error is discovered. In all other error cases (both syntactic and semantic), no result should be returned instead. The Query Parser must take this distinction into account, so that the correct output can be created.

If all validation checks pass, then a Query object containing the parsed information is returned.

This process is summarized by the activity diagram in Figure 3.5.3.1:

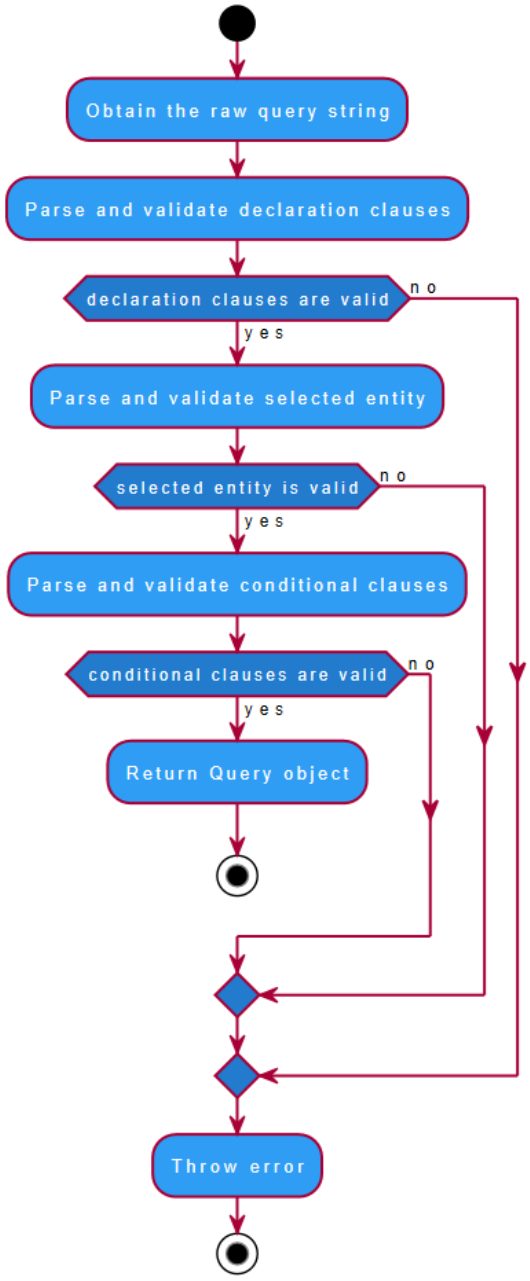

Figure 3.5.3.1: Overall activity diagram of the Query Parser parsing process

The main algorithm used for parsing and validation will be called **PQL regex validation**, which follows this three-stage process:

1. Extract the next term of the string, which is expected to be in a specific format.
2. Validate the syntactic and semantic correctness of the term via regexes.
3. Convert the parsed term into an object, and add it to the output Query object.

PQL regex validation will be used extensively in the three main parsing steps of the raw query string (Declaration clauses, Selected Entity, and Conditional clauses). These steps will each be elaborated upon in the following three sections.

3.5.3.3 Parsing of Declaration Clauses

Current raw query string parsing status:

```
assign a1; assign a2; constant c; print pn; variable v;
Select <a1, pn.varName> such that Uses(a1, v) pattern a2(v, _)
                            with c.value = pn.stmt# and 1 = 1
```

The method responsible for parsing and validating the declaration clauses of the query string is `ParseDeclarationClauses`. PQL regex validation ensures that the following rules are maintained:

1. Each declaration clause must fit the syntax of the PQL concrete grammar.
2. Each synonym can be declared only once.
3. Adjacent `NAME` or `INTEGER` terms must be separated by whitespace, or tokens that are neither `NAME` nor `INTEGER`.

The validation and parsing process can be seen in Figure 3.5.3.2 below.

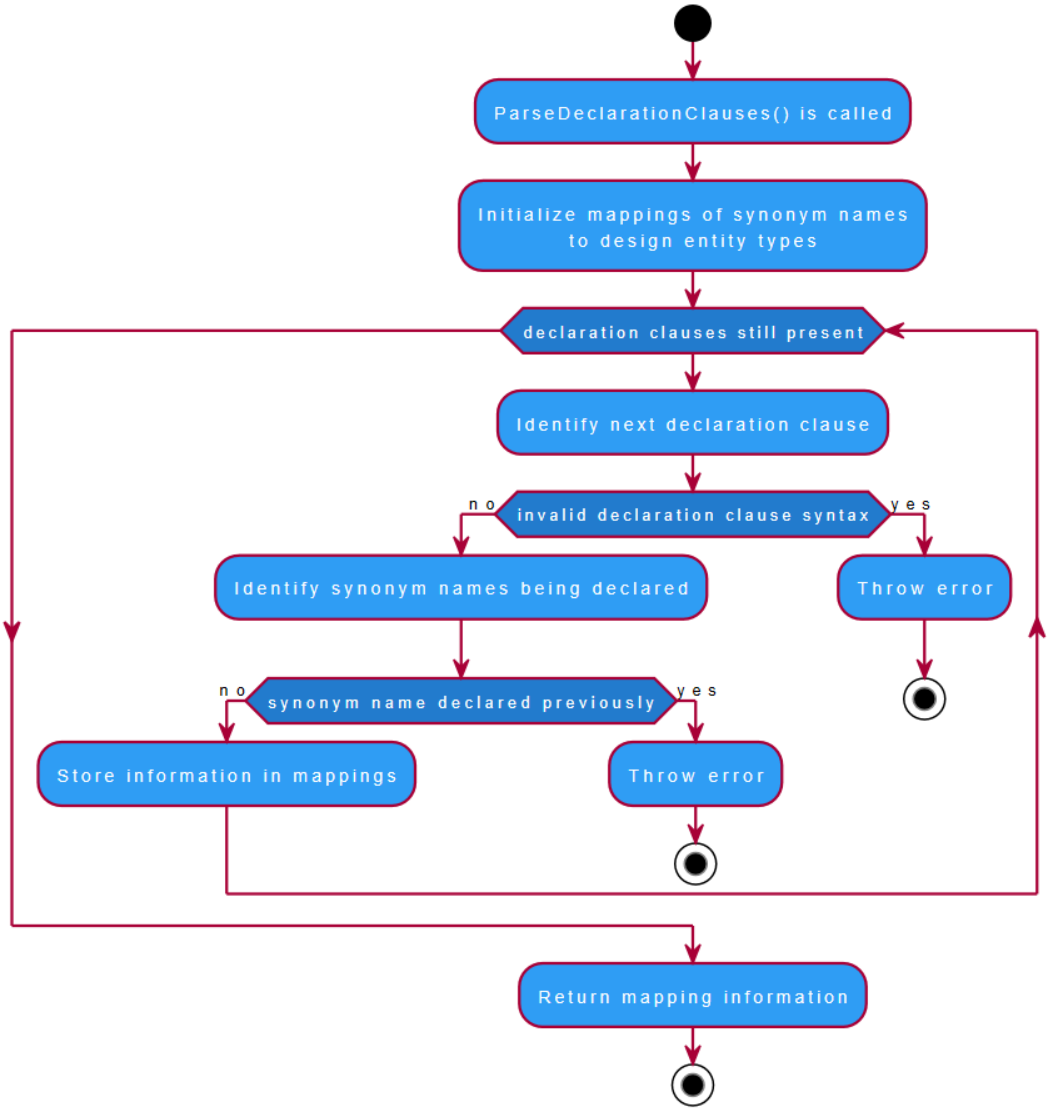

Figure 3.5.3.2: Activity diagram for the ParseDeclarationClauses method

Examples of Queries with Invalid Declaration Clauses

An example of a raw query string that would be considered syntactically invalid at this stage is the following:

```
stmt s Select BOOLEAN
```

In this case, the semicolon after `stmt s` is missing. Since the PQL concrete grammar requires that a semicolon be present at the end of each declaration clause, this violates the grammar and is therefore syntactically invalid.

An example of a raw query string that would be considered semantically invalid at this stage is the following:

```
print p; procedure p; Select BOOLEAN
```

This is because the same synonym (`p`) is declared twice. Since future references to `p` will ambiguously refer to two different design entity types, print and `procedure`, this is an error. However, since this raw query string does not violate the PQL concrete grammar, it is considered syntactically valid. This error is therefore classified as a semantic error.

Parsing of Example

The terms extracted from the raw query string at this stage are:

```
assign a1; assign a2; constant c; print pn; variable v;
```

The extraction process will validate that all design entity types (e.g. `assign`, `constant`) and synonyms (e.g. `a1`, `c`) are allowable under the PQL concrete grammar. In this case, `assign` and `constant` are accepted design entity type keywords, and `a1` and `c` are both valid under the definition of `synonym`.

It will then associate each synonym with its preceding design entity type. The Query Parser is able to recognize that both `a1` and `a2` belong to the same design entity type (`assign`), and will associate both accordingly.

Note that the raw query string might also have been declared as `assign a1, a2;` - both methods of declaration are considered equally valid, and result in the same parsed information. The Query Parser does not distinguish between the two.

The output of this method is a map of design entity types to their respective synonyms, as seen in Figure 3.5.3.3 below. In Figure 3.5.3.3, the boxes represent abstract data types, and the arrows represent an abstract mapping. For instance, the synonym `c` is mapped to the type `constant`.

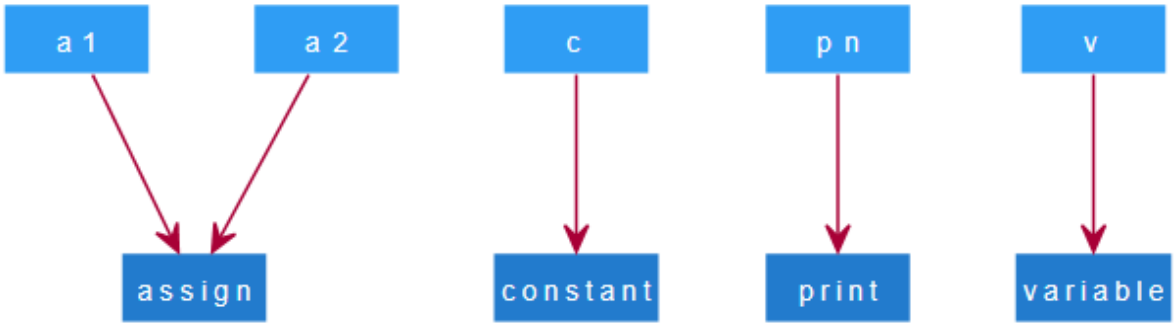

Figure 3.5.3.3: Status of synonym type mappings after ParseDeclarationClauses

The information contained within the mappings is preserved across the remaining steps of the query parsing process, where they are used in semantic validation. The next section will discuss the parsing and validation process of the Selected Entity.

3.5.3.4 Parsing of Selected Entity

Current raw query string parsing status:

```
assign a1; assign a2; constant c; print pn; variable v;
Select <a1, pn.varName> such that Uses(a1, v) pattern a2(v, _)
                            with c.value = pn.stmt# and 1 = 1
```

The Selected Entity is defined as the boolean, synonym, attribute, or tuple that immediately follows the ‘Select’ keyword in the Select clause. This Selected Entity ultimately determines which type of value to return from the Query Processor system.

The method responsible for parsing and validating the Selected Entity of the query string is `ParseSelectPhrase`. PQL regex validation ensures that the following rules are maintained:

1. The Selected Entity must fit the syntax of the PQL concrete grammar.
2. All synonyms present in the Selected Entity must have been previously declared.
3. Adjacent `NAME` or `INTEGER` terms must be separated by whitespace, or tokens that are neither `NAME` nor `INTEGER`.

The PQL regex validation for this method can be seen in Figure 3.5.3.4 below.

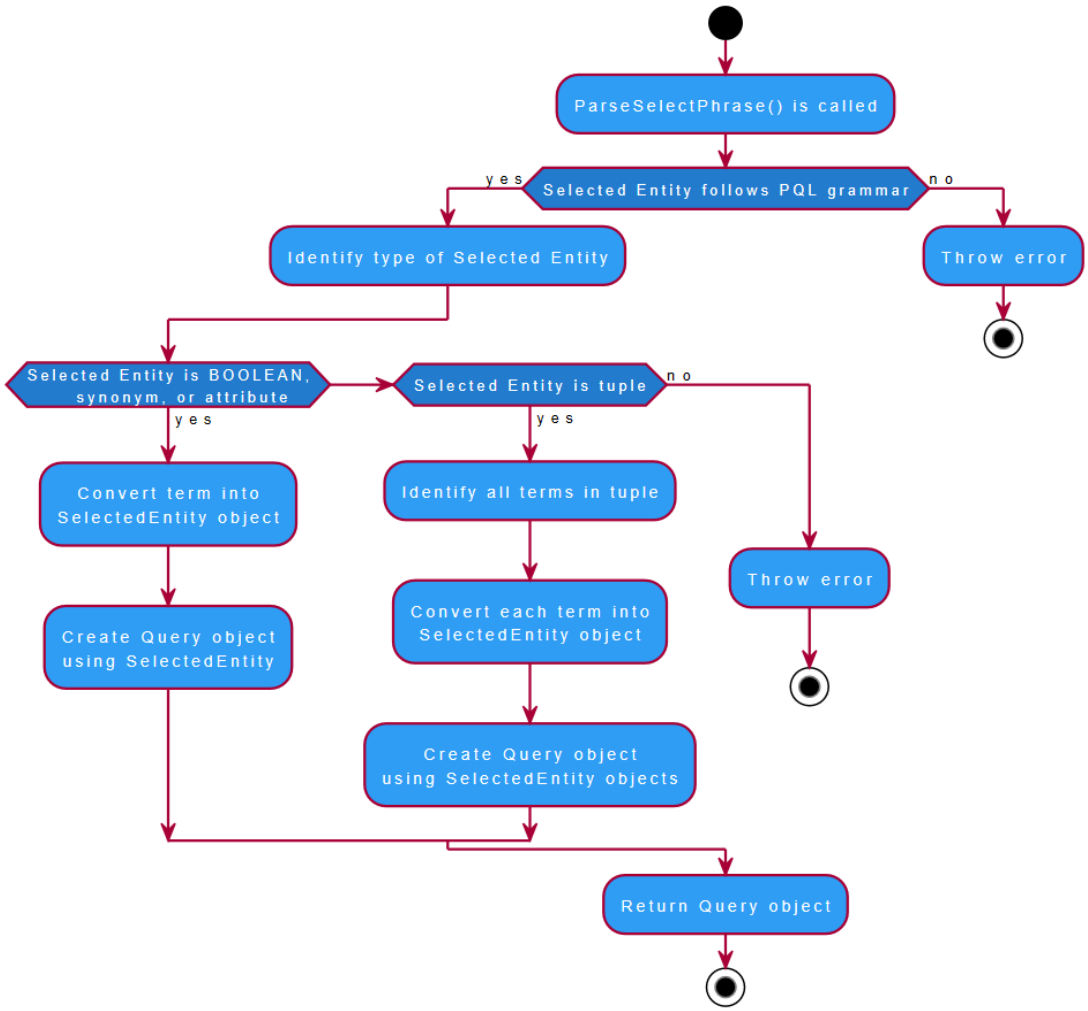

Figure 3.5.3.4: Activity diagram for the ParseSelectPhrase method

Based on the rules of the PQL grammar, the Selected Entity can be BOOLEAN, a declared synonym, an attribute, or a tuple containing some combination of synonyms and attributes.

Note that it is possible that a synonym can be declared with a name of BOOLEAN, for example in the query below:

```
stmt BOOLEAN; Select BOOLEAN such that Follows(1, BOOLEAN)
```

The Selected Entity is ambiguous in this case, because it is uncertain whether the True/False boolean is requested, or rather the synonym named BOOLEAN. In these cases, it is always assumed that the True/False boolean is requested, since the term 'BOOLEAN' is a keyword, and it is always possible for the user to name the synonym with another name, instead.

Therefore, the query evaluates `such that Follows(1, BOOLEAN)` as if the synonym named `BOOLEAN` was of type `stmt`, and returns True/False after the evaluation.

BOOLEAN, synonyms, and attributes consist only of a single term that needs to be parsed. However, in the case of a tuple, each term in the tuple must additionally be identified and parsed.

Parsing of Example

In the example query, the Selected Entity is a tuple, as follows:

```
Select <a1, pn.varName>
```

The first term in the Selected Entity is `a1`. This is associated with the design entity type of `assign`, as previously mapped in the `ParseDeclarationClauses` method.

The second term in the Selected Entity is `pn.varName`. This is recognized as an attribute with synonym `pn` and attribute type `varName`.

This information is used to initialize the Query object. Since the Selected Entity is a tuple, both terms in the tuple are encoded into the Query object as SelectedEntities, as seen in Figure 3.5.3.5 below.

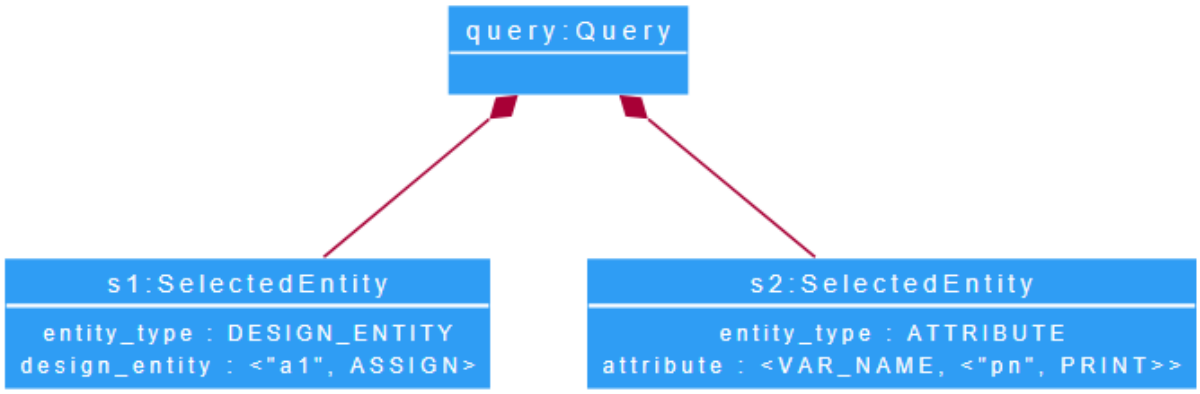

Figure 3.5.3.5: Current state of the Query object

The next section will discuss the parsing and validation of the conditional clauses of the query string, which will be used to populate more data into the Query object.

3.5.3.5 Parsing of Conditional Clauses

Current raw query string parsing status:

```
assign a1; assign a2; constant c; print pn; variable v;
Select <a1, pn.varName> such that Uses(a1, v) pattern a2(v, _)
                            with c.value = pn.stmt# and 1 = 1
```

The ‘Conditional Clauses’ are all clauses that impose a constraint on the synonyms and design entity types that have been declared. These clauses include ‘such that’, ‘pattern’ and ‘with’ clauses. In the AdvancedSPA requirements, any number of these clauses can occur in any order after the Selected Entity. Additionally, clauses of the same type may optionally instead be connected with the keyword `and`.

Each of these clauses must be evaluated independently by the Query Evaluator, so therefore the information from each of the clauses must be encapsulated in Clause objects.

The method responsible for parsing and validating the conditional clauses of the query string is `ParseConditionalClauses`. PQL regex validation ensures that the following rules are maintained:

1. Each conditional clause must fit the syntax of the PQL concrete grammar.
2. All synonyms present in each conditional clause must have been previously declared.
3. Adjacent `NAME` or `INTEGER` terms must be separated by whitespace, or tokens that are neither `NAME` nor `INTEGER`.

`ParseConditionalClauses` also serves as a wrapper `method` for the four helper methods that deal with the three types of conditional clause, and '`and`' keywords. These roles are fulfilled by the following methods:

1. `ParseSuchThatClause` handles 'such that' clauses
2. `ParsePatternClause` handles 'pattern' clauses
3. `ParseWithClause` handles 'with' clauses
4. `ReplaceKeywordAnd` handles '`and`' keywords

The parsing process of the entire `ParseConditionalClauses` method is further elaborated upon in Figure 3.5.3.6 below.

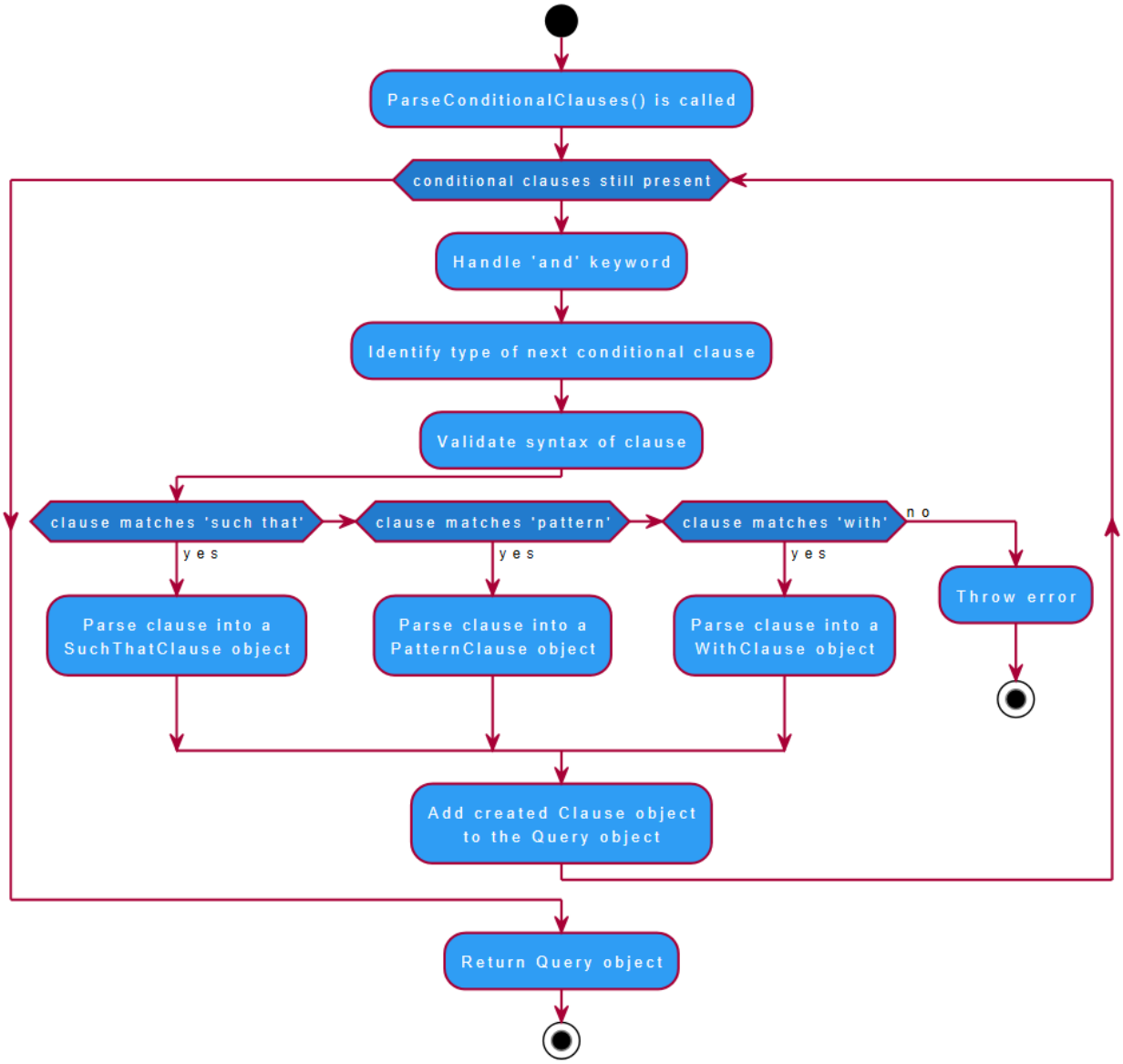

Figure 3.5.3.6: Activity diagram for the ParseConditionalClauses method

Parsing of Example

In the example, there are four conditional clauses:

- `such that Uses(a1, v)`
- `pattern a2(v, _)`
- `with c.value = pn.stmt#`

- `and 1 = 1`

Using PQL regex validation, the first three clauses are identified as ‘such that’, ‘pattern’, and ‘with’ clauses respectively. Some specific details to pay attention to are:

- Ensuring that `a2` in the ‘pattern’ clause is of type `assign`,
- Ensuring that the attributes `value` and `stmt#` in the ‘with’ clause are valid attribute types for the design entity types associated with the synonyms `c` and `pn`.

Note that some validation steps (e.g. ensuring that `c.value` and `pn.stmt#` return the same primitive type) are performed by the Query Evaluator instead.

The fourth clause (`and 1 = 1`) uses the `and` keyword, so `ReplaceKeywordAnd` is used to resolve the type of the conditional clause, which is ‘with’.

These four clauses are converted into Clause objects and inserted into the Query object. The final state of the Query object is shown in Figure 3.5.3.7 below:

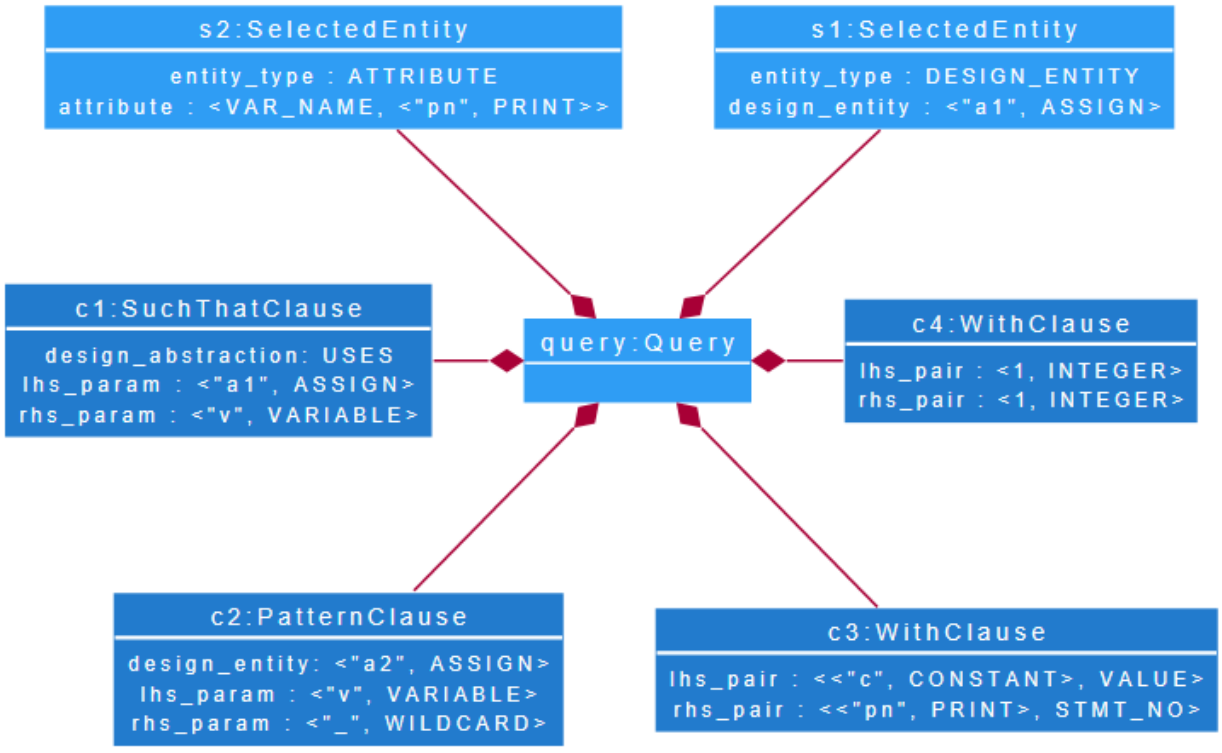

Figure 3.5.3.7: Final state of the Query object

This Query object is then returned from the Query Parser, and will be passed to the Query Evaluator to evaluate.

The previous three sections have discussed the specific motivations and implementations of the parsing of the query string. The following section will elaborate on some of the design decisions made when designing the Query Parser.

3.5.3.6 Design Decisions for Query Parser

Problem Statement: Choice of algorithm used to parse tokens from raw query string.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Types of tokens allowed in query string
Any amount of whitespace between almost any tokens of the input string. Multiple valid types of parameters in all clauses.

Evaluation Criteria:

1. Flexibility of Algorithm
2. Number of passes of Algorithm

Description of Possible Solutions:

Stream-based tokenization
Retrieve the next token by reading from a stringstream into a string buffer. Evaluate token when a terminating character is read. Identify parameters after sufficient tokens are evaluated.
String manipulation

Use regexes to evaluate the syntactic correctness of the following elements. Use string slicing and matching methods to identify specific parameters.

Evaluation of solutions

Flexibility of Algorithm	
Stream-based tokenization	Less flexible - difficult to encode different tokenization termination conditions for the various clauses.
String manipulation	More flexible - easier to match strings to regexes and slice strings along specific characters.

Number of passes of Algorithm	
Stream-based tokenization	Two passes - one for lexing the raw query string, and another for parsing the tokenized elements of the string.
String manipulation	Multiple passes - the next portion of the string must constantly be validated for syntax, and string manipulation involves creation of many intermediate strings.

Final Choice:

- String manipulation methods were chosen. The algorithm is less efficient, but is easier to implement, especially when new requirements are introduced over the course of the Iterations.
- Future extensions to this project may consider refactoring the Query Parser to use the stream-based tokenization approach instead, when efficiency is desired.

Problem Statement: Choice of regex used to syntactically validate query string.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Matchability to PQL grammar
The regex must allow all query strings that satisfy the PQL concrete grammar. The regex must disallow all query strings that do not satisfy the PQL concrete grammar.

Evaluation Criteria:

1. Size limit of Regex
2. Number of comparisons needed

Description of Possible Solutions:

Full-query regex validation
Build a complex regex from small components that exactly matches a valid query. At the start of the parsing process, match this regex to the query string to determine its syntactic validity.
Partial-query regex validation
Build multiple less-complex regexes from smaller components that each exactly match a specific part of a valid query (e.g. a single valid declaration clause). At the start of each section of parsing, validate the next part of the query with the equivalent regex to determine its syntactic validity.

Evaluation of solutions

Size limit of Regex	
Full-query regex validation	Much smaller - the regex match results in a stack overflow error with a query of several hundred characters (10-20 clauses).
Partial-query regex validation	Much larger - the regex match results in a stack overflow error only when a single element's length exceeds several hundred characters.

Number of comparisons needed	
Full-query regex validation	Only one comparison - the query string is only matched at the beginning of the parsing process.
Partial-query regex validation	Multiple comparisons - the query string is matched every time the next element needs to be identified.

Final Choice:

- Partial-query regex validation was chosen, even if it is less efficient.
- The full-query regex validation method is ideal for Iteration 1, since each query has a maximum of two clauses.
- However, with the clause limit removed in Iteration 2 and 3, the size limit of regexes becomes significant, so that a full-query regex validation method is no longer feasible.

3.5.4 Query Evaluator

3.5.4.1 Overview of Query Evaluator

The Query Evaluator accepts a Query object as well as a populated PKB as input and outputs a QueryResult object to be formatted by the Query Projector. The Query

Evaluator also accepts a boolean flag on whether queries and ResultTables should be preprocessed and merged by the Query Optimizer. The Query Evaluator is a static class, but it maintains a Database (not to be confused with the PKB) for each query, which contains the tables representing the query result space based on the evaluation of clauses.

3.5.4.2 Design of Query Evaluator

In the discussion of the Query Evaluator, the example query used is shared with the Query Parser:

```
assign a1; assign a2; constant c; variable v;

Select <a1, pn.varName> such that Uses(a1, v) pattern a2(v, _) with c.value = pn.stmt# and 1=1
```

The sample is used against the source code provided in Figure 3.5.4.1. In this section, it is assumed that clauses are evaluated in the order as given in the query above from left to right.

Result Table

The Query Evaluator mainly operates and stores intermediary data in a custom data structure, the ResultTable class. The ResultTable class simply contains a map of synonyms to vectors. Each entry in the map represents a Column, with the synonym as a header and a vector of data represented by the synonym. The row on which the data is on is represented by its index in the vector. Each row stores either a statement index or a name. For example, the evaluation of Uses(a1, v) on the following program would give rise to the ResultTable as represented in Figure 3.5.4.1.

```
01 a = b + c

02 b = b + c

03 b = x + y

04 b = 5

05 print b
```

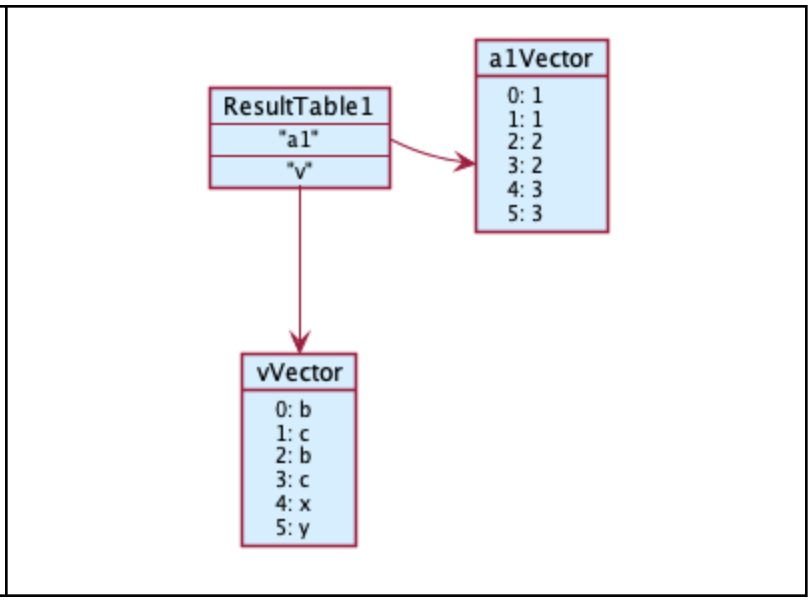

Figure 3.5.4.1 ResultTable returned from evaluation of Uses(a1, v)

The Query Evaluator stores a collection of ResultTables in a vector, forming the Database that may be used while evaluating clauses. The database is only used for the evaluation of a single query and is deleted after the evaluation of a query.

Merging of Result Tables

The Result Table supports merge operations (merging of two different tables) in two different ways, via cross product or inner join.

Given any result table, if the two tables have any intersecting columns at all, merging would result in an inner join of the table. That is, the table will be merged on it's intersecting columns, keeping only rows with intersecting values. For example, an inner join of the following two tables happen as follows:

a	v		a	w		a	v	w
1	x	Inner join ↔	2	4	Result →	2	y	4
2	y		3	5		3	y	5
3	y							

If there are no intersecting columns, then an $O(n^2)$ algorithm to cross product the two tables will be performed.

Overall Evaluation of Query

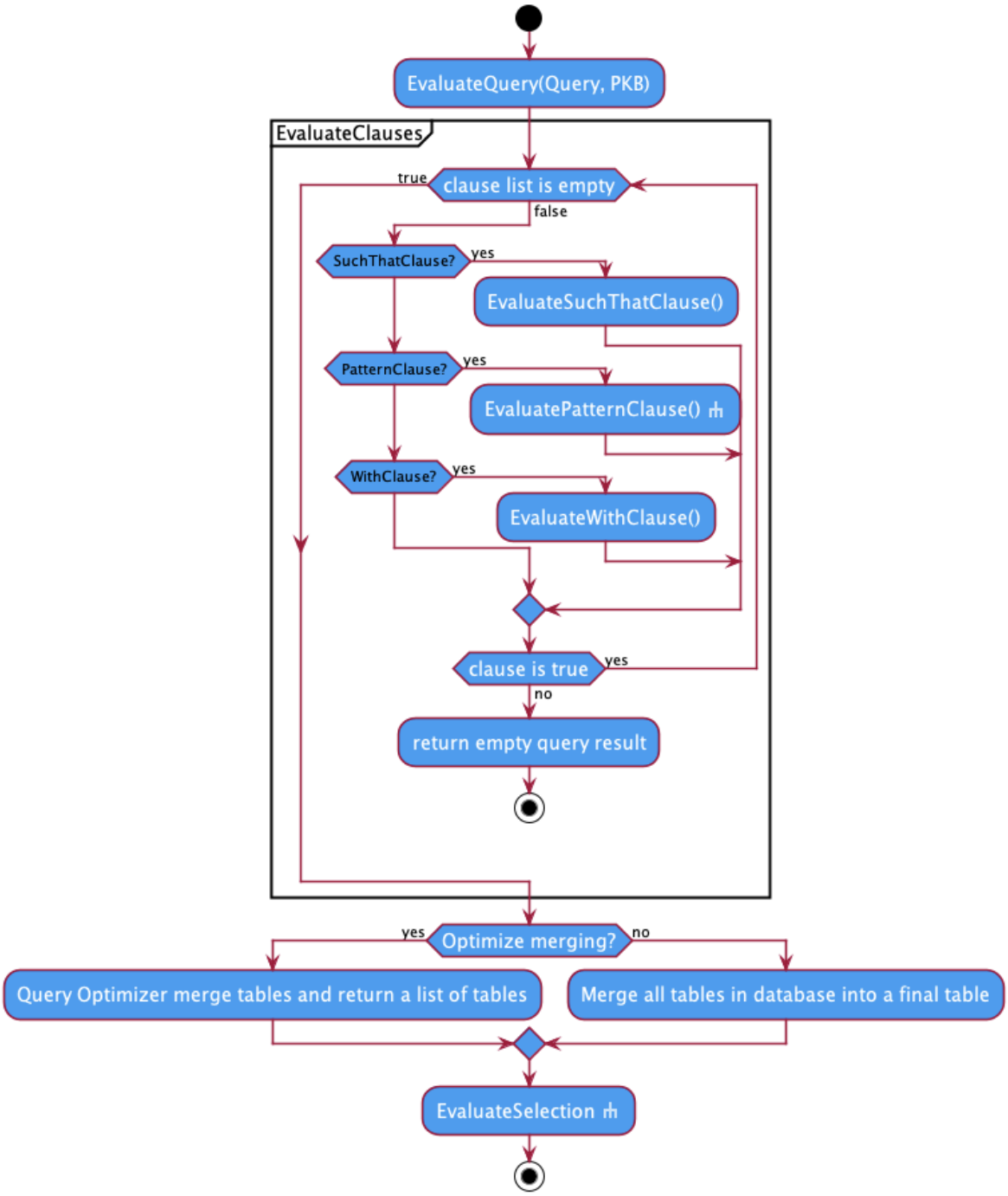

Figure 3.5.4.2 Overall execution of the EvaluateQuery function

The Query Evaluator evaluates clauses indiscriminately in the order in which they are received. The evaluation of any clause to false immediately returns an empty QueryResult (or FALSE for BOOLEAN) to the Query Projector. Note that in Iteration 3, the query is preprocessed by the Query Optimizer so that clauses can be evaluated in a more efficient order. This will be further elaborated on in Section 3.5.5.

After the evaluation of a true clause, the results of the evaluation are stored in a ResultTable and added to the Database. Note that after the evaluation of a clause, the ResultTable is **not merged** with previous ResultTables.

Instead, merging of the tables will only occur at the end of the evaluation, after all clauses have evaluated to "true" (for the justification, refer to design decisions for the Query Evaluator). This does not mean that the query is true, since the reduction of space of one clause will propagate to the previous clauses. Hence, relevant tables will have to be merged. If the Query Optimizer is toggled on, the Query Optimizer will perform the merging and return a list of merged tables instead. If not, the Query Evaluator merges all existing ResultTables into a single table. After the merging of all tables, it is determined whether the query evaluates to true as a whole based on:

1. If the final table(s) have no columns, i.e no synonyms were evaluated in the course of the evaluation of the query, then the query evaluates to true as a whole, since all clauses evaluated to true.
2. If the final table(s) have columns, synonyms were evaluated in the course of the evaluation query. In this case, it can be determined that the query evaluates to true if there are **rows** in all tables, which would mean that there exists a set of synonyms that fulfill the given relationships. Else, the query evaluates to false.

Finally, relevant columns from the final tables are selected based on the SelectedEntity, and converted to a QueryResult object to return to the Query Projector.

Evaluation of Clauses

In general, the heuristic for the evaluation of clauses is as such:

For any two parameters (left hand side and right hand side) that are being evaluated, a ResultTable is first created based on the following steps.

1. If both parameters are synonyms, iterate through the Database to check if there is an existing ResultTable that contains **both** synonyms. If so, return the latest ResultTable that fulfills the condition, and remove that table from the Database. This is so as to avoid any unnecessary cross product between columns, since both synonyms have already been merged and evaluated previously. The ResultTable with the updated result space will be added back to the Database after evaluation.
2. If a parameter is an index or a string, convert it to a Column with a single row.
3. If a parameter is a synonym, iterate through the Database (from the back to get the latest one) to check if it exists in any ResultTable. If yes, retrieve the Column for that synonym and remove any duplicate elements. If not, retrieve the design entity table from the PKB and convert to a Column.
4. Cross product both Columns to form a ResultTable.

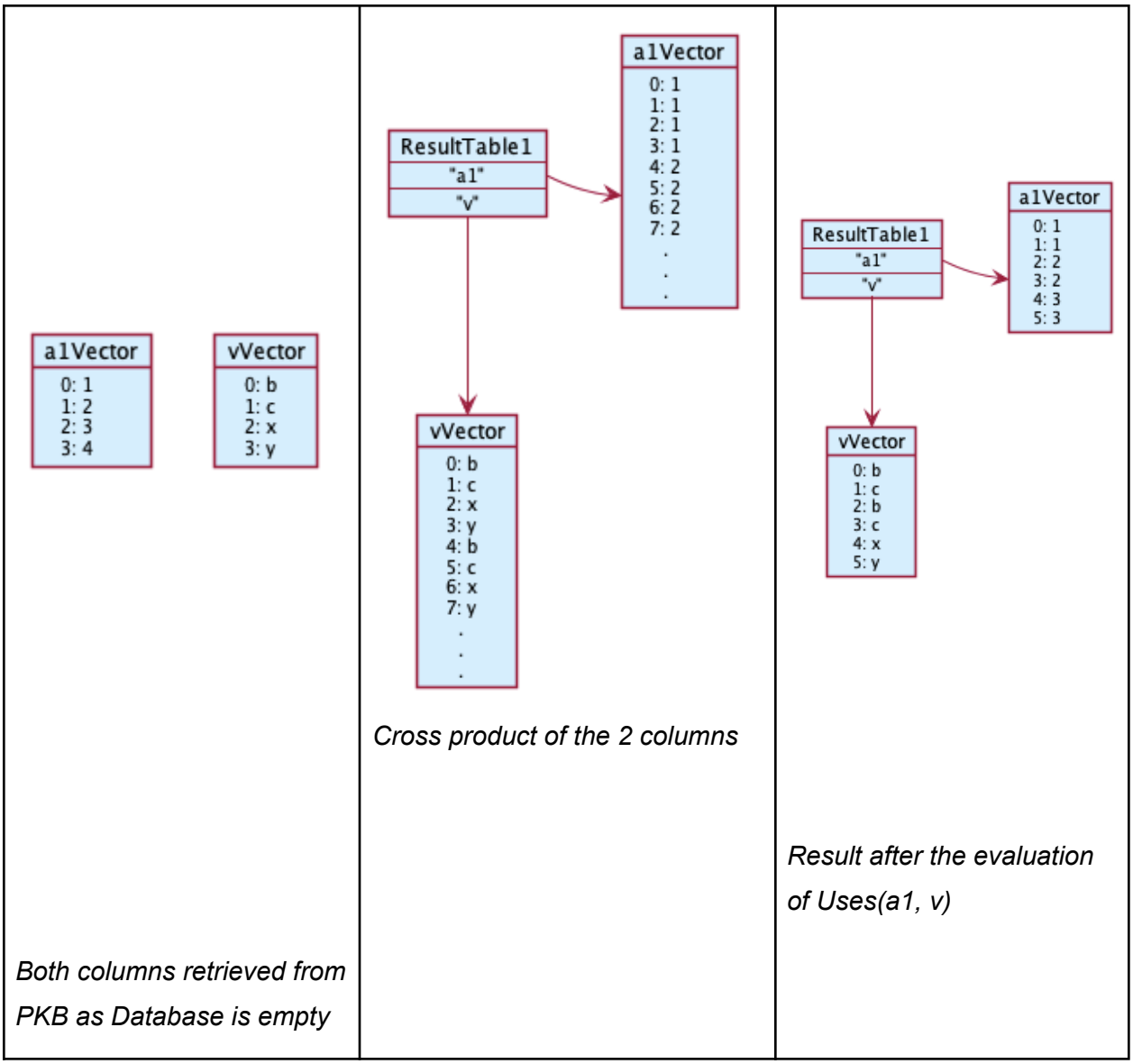

Figure 3.5.4.3 ResultTables in the evaluation of Uses(a1, v)

The ResultTable represents all **current possible** permutations of the two parameters. Then, each row of the ResultTable is evaluated against the relationship or Design Abstraction that is required by calling the API methods exposed by the PKB. If any given row does not fulfill the relationship, it is then removed from the ResultTable. After iterating through all rows, the resultant ResultTable contains possible solutions to the given clause. Note that these solutions are not final since they can be further reduced by subsequent clauses during the merging of tables.

Figure 3.5.4.3 shows the intermediate result tables for the evaluation of `Uses(a1, v)` based on the source code given in Figure 3.5.4.1. The evaluation illustrates Steps 3 and 4 of the algorithm as no tables are currently present in the Database, hence the columns are retrieved from the PKB and a cross product is performed.

All three types of clauses are evaluated in the same way, with minor differences in the evaluation of each row.

For Such That Clauses, the PKB API for the relevant Design Abstraction will be called to check the validity of the row. For example, with reference to Figure 3.5.4.3, `IsUses(1, "x")` will be called while evaluating the third row of the table (index 2). The PKB returns false in this case and the row is subsequently removed from the result table.

For With Clauses, each row needs to be converted to its attribute before checking for equality. In the example, given `with pn.stmt# = c.value`, the print design entity and constant design entity will be retrieved from the PKB. If these synonyms had been previously evaluated by any clauses, they would be retrieved from the Database instead. Since both print and constant are already in the form of statement numbers and values respectively, nothing further needs to be done. However, if it had been pn.varName instead, before the evaluation of each row, the PKB API `GetPrintVarName(int)` has to be called to get the variable name used in the particular print statement. Finally, the equality of every pair of print statements and constant values is checked and if they are not equal, the row is removed from the ResultTable. From the example, `with pn.stmt# = c.value` evaluates to true with (5, 5) as the only pair of answers based on the source code in Figure 3.5.4.1. For values that are not design entities, e.g `with 1=1`, they are simply evaluated as a normal equality.

Pattern Clauses differ slightly as it is a two step evaluation process (for assign pattern clauses). In assign pattern clauses, it is noted that the evaluation of the LHS param is essentially synonymous to `Modifies (a2, v)`. Hence, that is evaluated first. After

which, all assignment statements containing the pattern expression on the RHS are retrieved from the PKB. Subsequently, all rows in the ResultTable from `Modifies (a2, v)` whose assignment statement does not contain the pattern expression are removed from the ResultTable. For conditional pattern clauses (while and if), the first step is not necessary. Given `pattern w(v, _)`, a result table for containing w and v is created as though `w` is the LHS param and `v` is the RHS param. For each while statement in the table, the PKB API to get all variables used in the conditional of that particular statement is called. If the corresponding variable in the same row is found in the set returned from the PKB, the row is not deleted.

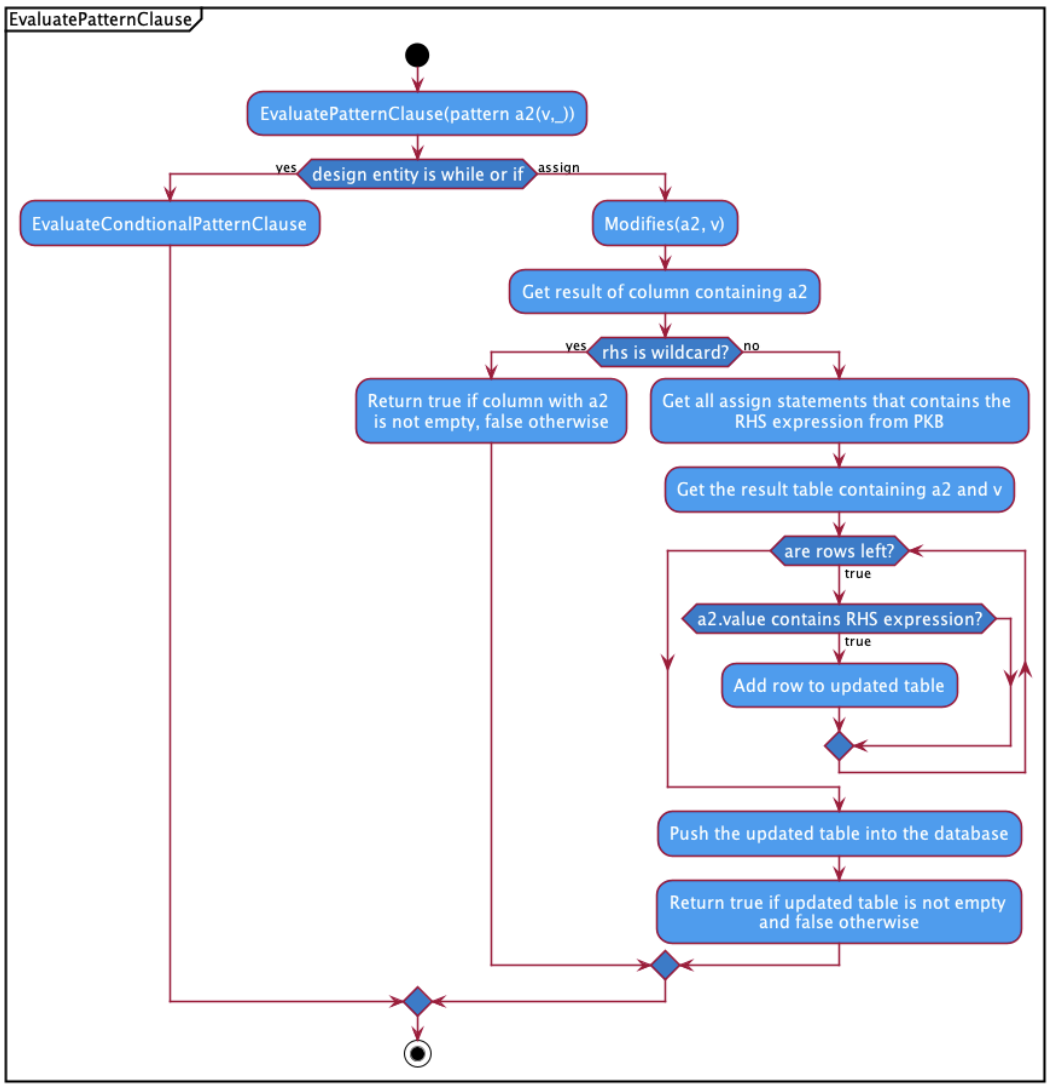

Figure 3.5.4.4 Evaluation of pattern a2(v, _)

Merging Tables

After evaluation of all clauses, assuming that the Query Optimizer does not handle the merging, ResultTables in the database are merged into a single final table. Figure 3.4.5.5 illustrates how this is done as per the given example. Note that the result space of a1 is reduced from [1, 2, 3] to [1, 2] by merging the table of `pattern(a2, v_)`. Additionally, a cross product is done between the results of Uses(a1, v) pattern a2(v,_) and the results of `with pn.stmt# = c.value`.

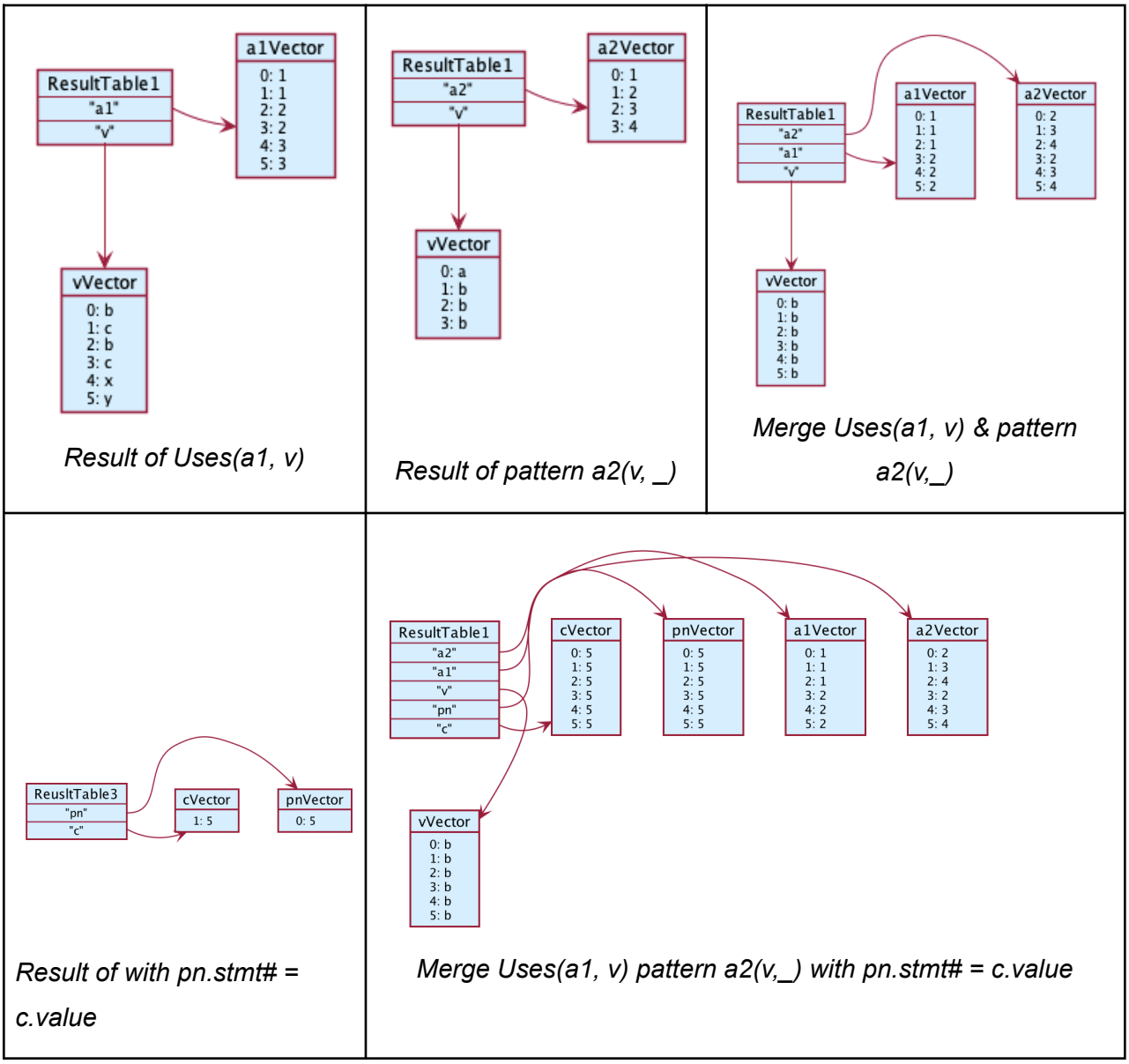

Figure 3.5.4.5 Merging of ResultTables into a single table

However, had Query Optimizer handled the merging instead, it would have avoided the cross product since it did not change the result space of any of the synonyms, and returned two ResultTables instead:

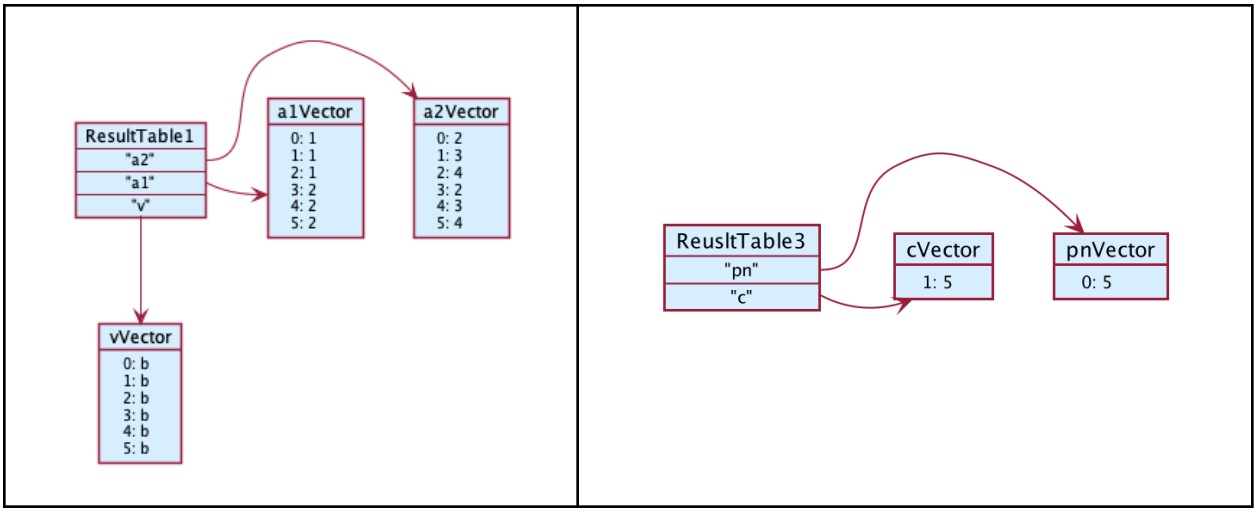

Figure 3.5.4.6 Final result of merging by Query Optimizer

Selection

When the query is determined to be true, the selection of the query is evaluated. If the selection is a BOOLEAN, the Query Evaluator simply outputs TRUE.

Synonyms in general will first be searched for in the list of final ResultTables. If it exists, the synonym column will be returned. If not, the table for that design entity type will be retrieved from the PKB. If the attribute of a synonym is selected (e.g `pn.varName`), print statements are retrieved similar to that of synonyms, then each statement is converted to its corresponding attribute by calling relevant methods from the PKB.

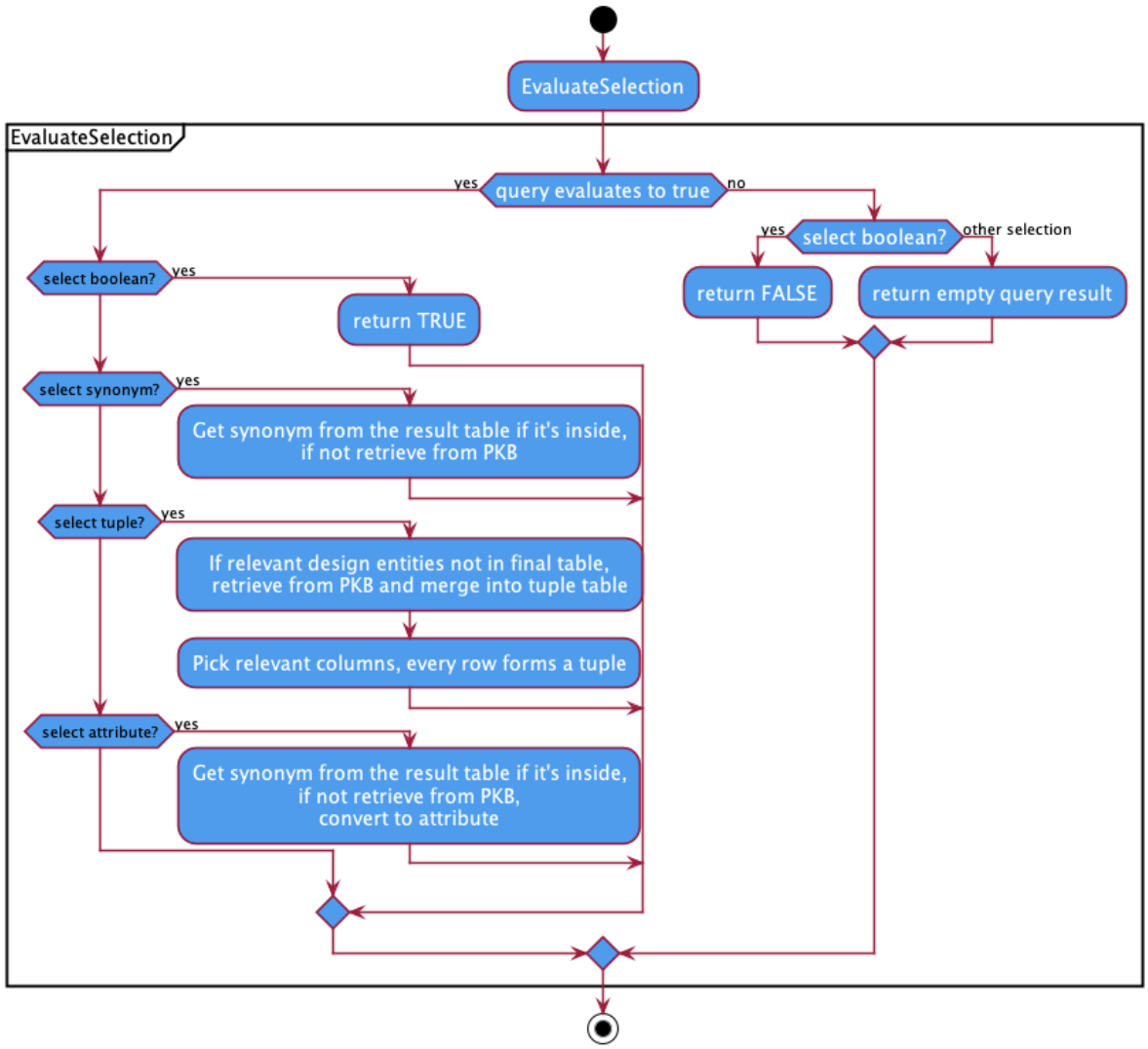

Figure 3.5.4.7 Activity Diagram for EvaluateSelection

Last but not least, selection for tuples will construct a ResultTable with the required synonym columns. It iterates through each ResultTable in the final Database and retrieves the relevant columns, and performs a cross product with any existing columns in the required columns. For example, given the two tables in Figure 3.5.4.6 and `Select <a1, pn.varName>`, the `a1` Column will be retrieved from the first table, and `pn` Column from the second table. Since the selected attribute is varName, the PKB API to `GetPrintVarName` is called for each row in pn. These two columns are then cross

joined to give the final tuples of <1, 5> and <2, 5>. Note that any duplicate rows are duly removed. Finally, if there are any selected synonyms not found in any of the ResultTables will be retrieved from the PKB and will be merged into the required columns with a cross product as well. Finally, the Query Evaluator parses each row into a vector of strings, representing each result for the tuple.

3.5.4.3 Interactions with the PKB

The table below summarizes all APIs used by the Query Evaluator

Purpose	**APIs**
To ascertain the validity of any Design Abstraction relationships between two values in the evaluation of SuchThatClauses	`IsFollows(int s1, int s2)`, `IsUses(int s1, string var)`, and all other IsXX APIs for each Design Abstraction. Details can be found in Appendix 8.1
To obtain the members of each design entity in the source code	`GetAllStmts, GetAllVariables`, and all other GetAllXX APIs for each Design Entity. Details can be found in Appendix 8.1

To convert design entities to their corresponding attribute	`GetCallsProcName`
	`GetPrintVarName`
	`GetReadVarName`
	*All other attribute types correspond to the original design entity type

To evaluate patten expressions in assign pattern clauses	`GetAllAssignStmtsThatContains(TokenList pattern_expr)` `GetAllAssignStmtsThatMatches(TokenList pattern_expr)`
To evaluate conditional pattern	`GetVariablesUsedByWhileStmt(int stmt)`

clauses	`GetVariablesUsedByIfStmt(int stmt)` * These APIs only return variables used in the conditional expression.

3.5.4.4 Validation done in Query Evaluator

Validation done in the Query Evaluator

While the Query Parser validates syntactic errors in the queries, it was not designed to pick up on all semantic errors. Hence, further validation on the queries is done before the evaluation of the clauses.

These are the validation rules implemented in the Query Evaluator:

For Follows, Follows*, Parent and Parent*, Next, Next*, NextBip, NextBip*, Affects, Affects*, AffectsBip, AffectsBip*:

1. LHS param and RHS param must either contain a statement index, a wildcard or a DesignEntity that is **not** of type procedure, variable or constant.

For Uses and Modifies:

2. LHS param must be either a statement index, name or a DesignEntity that is **not** of type procedure, variable or constant. Wildcards are not allowed here.
3. RHS param must be either a name, a wildcard or a DesignEntity that is of type variable.

For Calls and Calls*:

4. LHS param and RHS param must either be a name, wildcard or a DesignEntity of type procedure

For pattern clauses:

5. DesignEntity involved must be of type **assign, while or if**
6. LHS param must be either a string, a wildcard or a DesignEntity of type variable.

7. RHS param must be a wildcard for **while and if**
8. RHS param must be a wildcard or Pattern Expression for **assign**

For with clauses:

9. The attribute attached to each DesignEntity must be valid (e.g only constants can have type VALUE, assign cannot have VAR_NAME etc.)

Selection:

10. Tuples cannot contain BOOLEANs
11. Selected attributes attached to each DesignEntity must be valid

3.5.4.5 Design Decisions for Query Evaluator

Problem Statement: Algorithm for merging and storing of intermediate result tables

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Correctness
The final result of evaluation must be correct. Merging of tables is necessary to some extent as future clauses can affect the result space of previous clauses.

Evaluation Criteria:

1. Efficiency of evaluation
2. Efficiency of merging
3. Flexibility of optimization

Description of Possible Solutions

Single intermediate table
Store a single intermediate table. During the evaluation of each clause, get the required synonyms from the table if available. After evaluation, merge the resultant table from the clause back into the intermediate table.
Store a list of Result Tables for each clause
For each clause, search through the list of ResultTables to find the required synonyms if available and merge them if necessary. After evaluation, do not merge. Merging occurs after all clauses have been evaluated.

Evaluation of Solutions

Efficiency of Evaluation	
Single intermediate table	Given a clause with many synonyms, this is extremely inefficient. This is because merging synonyms result in an exponentially growing table. During the evaluation of a clause, a column may contain many unnecessary rows that are repeatedly evaluated. After testing, this solution was unable to handle more than 5 clauses or so without a significant slowdown. However, retrieving previously evaluated synonyms occurs in O(1) time.
Store a list of Result Tables for each clause	Evaluation here is slightly more efficient. Each intermediate table will have a maximum of two columns. However, it is noted that when retrieving a previously evaluated synonym, the space of the synonym might not have been reduced to the smallest possible size. For example, in the query previously used `Uses(a1, v) pattern a2(v,_)`, the evaluation of pattern clause reduces the

	size of a1. This is not reflected in the intermediate steps. If a third clause were to use a1 again, the result space of a1 is not minimized. There is also a slightly greater overhead in retrieving previously evaluated synonyms since there is a need to iterate through result tables.

Efficiency of merging	
Single intermediate table	In general, the efficiency of merging the tables are the same, since the merging algorithm used is the same, and each clause requires the merging of a smaller table (evaluated by the clause) to a large table. The only difference is that the second solution merges the table at the very end, after evaluation of all clauses. It is possible to slightly optimize the merging process. The inner join algorithm makes use of a hash join, and by building the hash on the smaller table, the merge can be slightly more efficient.
Store a list of Result Tables for each clause	

Flexibility for Optimization	
Single intermediate table	This is inflexible to optimization in terms of merging. However, optimization can still occur by ordering clauses in a way such that clauses with overlapping synonyms are evaluated first to reduce the size of the intermediate table.
Store a list of Result Tables for each clause	Greater flexibility in optimization. Two points of optimization are available, firstly, clauses can be ordered to evaluate clauses with overlapping synonyms together. Secondly, it becomes possible to avoid merging of all intermediate tables. Clauses in two different groups (no overlapping synonyms and no connection via

	intermediate clauses) in fact do not have to be merged since they will not affect each other's result space.

Final Choice: Store a list of Result Tables. This allows greater flexibility and extension for the Query Optimization, which will be the component that significantly improves the runtime efficiency of merging and evaluation.

3.5.5 Query Optimizer

3.5.5.1 Overview of the Query Optimizer

The Query Optimizer is a separate static component from the Query Evaluator, and only the Query Evaluator is dependent on the Query Optimizer. No other components have any dependencies on the Query Optimizer. The Query Optimizer exposes two APIs that each contain several smaller optimizations:

API	Description	Optimizations
`OptimizeQuery`	Preprocesses clauses before the Query Evaluator evaluates the clauses	1. Removing duplicate clauses 2. Sorting clauses by design abstraction and number of synonyms
`OptimizeMerging`	Merge the list of tables in the Query Evaluator database based on overlapping synonyms, hence replacing the trivial merging algorithm in the Query Evaluator	3. Merging ResultTables by overlapping synonyms using BFS 4. Sorting ResultTables before any merging

Each optimization can be toggled on and off from the Query Evaluator. It is possible to run queries without any optimization, or run queries with only specific optimizations.

In the next two sections, each of the optimizations will be discussed, and in Section 3.5.5.4, the testing for the optimizations.

3.5.5.2 Optimizations of the Query

In this section, any preprocessing of the Query before the object is passed onto the Query Evaluator is discussed. In this section, a different example query than the query used in the discussion of previous Query Processor components will be used:

```
assign a; constant c; stmt s1, s2;
Select BOOLEAN such that Next*(s1, s2) and Next*(s1, s2) and
Follows(s1, s2) with 1=1 pattern a(“x”, _“a*b”_) with a.stmt# =
c.value such that Parent(s2, _) with a.stmt# = c.value
```

For the sake of brevity, following instances of this query example will omit the declaration clauses. Additionally, although the query is displayed in free text, it is actually abstracted in a Query object. It should be noted that the Query Optimizer is **not** handling the raw query string, but rather the abstracted information in the Query object.

Removing Duplicate Clauses

Since the order of evaluation of clauses does not matter and all clauses are connected with a commutative AND operation, re-evaluating clauses that are exactly the same will not yield different results.

Hence the example query can be shortened to:

```
Select BOOLEAN such that Next*(s1, s2) and Follows(s1, s2) with 1=1
pattern a(“x”, _“a*b”_) with a.stmt# = c.value such that Parent(s2, _)
```

Since Clauses are stored in a vector, searching through the list for all repeated clauses requires $O(C^2)$ time, where C is the number of clauses. This is considered to be a small trade-off as compared to evaluating hundreds or even thousands of additional entries for each repeated clause, especially when queries are extremely unlikely to have more than 100 clauses in a typical use case. Hence, all duplicate clauses in each query are removed. Clauses are considered to be duplicates when they are of the same type (SuchThat, Pattern or With), and have the same left hand side and right hand side

parameters. SuchThatClauses need to have the same DesignAbstraction, while PatternClauses must have the same DesignEntity (`assign`, `while` or `if`).

Sorting of Clauses

Based on the implementation in the Query Evaluator, where the result space of previously evaluated synonyms will be utilised in the next clause's evaluation, clauses should ideally be evaluated starting from those that would yield the smallest table size. It is also ideal to evaluate that have a small result space first as they are more likely to result in the query to return false, hence terminating the evaluation in a fail-fast manner. Clauses with a small result space also evaluate faster, and in general, clauses that evaluate faster should be evaluated first. In the event that these clauses result in the query returning false, the more costly clauses can avoid evaluation.

Clauses are hence sorted via these three criterias, in the order: number of synonyms/wildcard, type of clause, type of design abstraction (only for SuchThatClauses).

Number of synonyms/wildcards

Clauses are first sorted by assigning a value to each parameter that it owns. The higher the value, the most costly the evaluation is estimated to be. The values are assigned as such:

Integer or string	0
Pattern expression	0
Synonym (for a design entity, includes assign/while/ifs for pattern clauses)	1
Wildcards	1

Having a definitive value such as an integer or string is likely to shrink the result size of any accompanying synonym significantly, and clauses with both such params evaluate in O(1) time (e.g `with 1 = 1),` hence they are evaluated first.

Synonyms include any synonym in the clause, including those attached to the start of pattern clauses (assign/while/if).

Wildcards represent a design entity retrieved straight from the PKB, and could be arguably more expensive than synonyms to evaluate. For example, `Follows(s1, _)` could be more costly than `Follows(s1, s2)` if s2 had previously been evaluated by another clause. However, the resulting space of `Follows(s1, _)` only contains s1 and is likely to be smaller than that of `Follows(s1, s2)`, since the table containing s1 and s2 needs to hold all combinations of s1 and s2. In addition, clauses with two wildcards employ a fail fast optimisation as discussed in Section 3.5.4.3. However, the evaluation is still bounded by $O(n^2)$. To avoid complicating things, synonyms and wildcards are allocated the same score.

The example query is then sorted as such:

```
Select BOOLEAN with 1=1 pattern a(“x”, _“a*b”_) such that Next*(s1,
s2) and Follows(s1, s2) with a.stmt# = c.value such that Parent(s2, _)
```

The with clause had a score of 0, while the pattern clause had one design entity “a” hence it was given a score of 1. The remaining SuchThatClauses each have two synonyms and are hence not sorted.

Type of clause

Within groups of clauses with the same score, clauses are then sorted by the type of clause, in the order: WithClause < PatternClause < SuchThatClause.

This was determined based on the estimated size of their result table space, and further sorts the example query:

`Select BOOLEAN with 1=1 pattern a(“x”, _“a*b”_) with a.stmt# = c.value such that Next*(s1, s2) and Follows(s1, s2) and Parent(s2, _)`

WithClauses are the fastest to evaluate as they mostly require a similar equality comparison. Additionally, the table size of with clauses is bounded by *min(number of values for LHS param, number of values for RHS param)*. For example, the resultant table size of `with s.stmt# = a.stmt#` is bounded by the number of statements in `a` or `s`, whichever has a smaller result space.

For PatternClauses, `pattern a(v, _)` is essentially the same as `Modifies(a, v)`. Having any expression on the right hand side shrinks the result space further. On the other hand, `pattern w(v, _)` and `pattern ifs(v _, _)` are strictly smaller than `Uses(w, v)` and `Uses(ifs, v)` respectively. Hence, PatternClauses are evaluated before SuchThatClauses.

Design Abstraction

SuchThatClauses are then further sorted based on the Design Abstraction. This optimizes the query significantly when there are design abstractions of varying sizes that are evaluated on similar synonyms. For example, given the example query that contains `Follows(s1, s2) and Next*(s1, s2)`, if the Next* clause were to be evaluated first, it would store a huge result table in the database, and Follows will be evaluated based on that table. However, if Follows is evaluated first, Next* will be evaluated on a significantly reduced space, hence saving some time.

Some simple heuristics were used to rank the Design Abstractions in increasing order. For example, these are some heuristics that is known to hold true in general: any of the transitive abstractions ≥ its associated non-transitive abstraction (e.g `Follows*` ≥ Follows). Also, `Next` ≥ `Follows` and `Next*` ≥ `Follows*` as all statements in the same stmtLst will share a path in the Control Flow Graph as well. In general, `Uses` ≥ `Modifies`, although this does not always hold true if there are many assignment statements that use constant values instead of variables. Abstractions like `Affects` and `Calls` which are restricted by the number of assignment statements and call statements

respectively are also likely to be smaller than others. To establish the ranking amongst all the design abstractions, some tests were done as elaborated on in Section 3.5.5.4. The final ranking is as follows:

```
Calls < Calls* < Affects < Follows < AffectsBip < Parent < Affects*
Next < NextBip < Follows* < Parent* < Modifies < Uses < AffectsBip* <
Next* < NextBip*
```

The clauses are then sorted from smallest to largest, hence the example query becomes:

```
Select BOOLEAN with 1=1 pattern a(“x”, _“a*b”_) with a.stmt# = c.value
such that Follows(s1, s2) and Parent(s2, _) and Next*(s1, s2)
```

3.5.5.3 Optimizations of Table Merging

After the Query Evaluator evaluates each clause, a list of ResultTables is obtained. The example query gives a list of four ResultTables with headers:

```
(a), (a, c), (s1, s2), (s2)
```

**Note that there is only one (s1, s2) table as repeated evaluation of identical parameters will update the previous table as explained in the Design of the Query Evaluator under Evaluation of Clauses.*

While the Query Evaluator trivially merges all ResultTables, the Query Optimizer only merges tables if it is absolutely necessary, and performs small optimisations on the merging. The Query Optimizer then returns a list of ResultTables to the Query Evaluator for selection after merging necessary ResultTables. Note that this step replaces the trivial merge in the Query Evaluator, and can be toggled on and off by a flag.

Merging Result Tables by groups

The Query Optimizer groups ResultTables together based on the overlapping synonyms, and only merges tables together if they have any overlapping synonyms. This avoids any unnecessary cross products of two tables.

This can be modelled as a graph problem. Each ResultTable is a node, and there exists an edge between two nodes if the ResultTable contains any overlapping synonyms. For example, there is an edge between ResultTables that contain headers `(a, w)` and `(w, s)`, but no edge between ResultTables that contain headers `(a, w)` and `(a1, w1)`.

After constructing the graph, a Breadth-First-Search (BFS) is done on each node that has not been merged, and all nodes (ResultTable) reachable by the node will be merged into a single result table. Neighbours of every node are guaranteed to have at least one overlapping synonym, hence using the cheaper inner join algorithm instead of a cross join. Hence a BFS ensures that all merges will utilise the inner join.

There is no need to merge ResultTables that do not have a path (direct or indirect) between them, as the result space will not be altered, even by propagation. Hence, nodes in separate groups will be merged into separate ResultTables.

Based on the example query, `(a)` and `(a, c)` will be merged into one ResultTable, and `(s1, s2)` and `(s2)` into another.

Sorting tables by size before BFS

This is a smaller optimization that can be toggled before any BFS is done. In the case that the prior sorting of clauses was inaccurate in estimating the ResultTable size of each clause, there might be a case where larger tables are merged first, resulting in a larger intermediate ResultTable. Hence, it might be more efficient to sort the tables before performing any BFS, hence any BFS will start from the smallest possible ResultTable.

3.5.5.4 Testing of Query Optimizer

Sorting of Design Abstractions

In the previous section, it was mentioned that SuchThatClauses were sorted by the estimated size of their result space based on the DesignAbstractions. While there are some simple heuristics that were applied, to obtain a definitive ranking, a random

source code generator was used to generate about 40 different source codes , and a query was run to find the table sizes of all possible Design Abstractions.

The average number of rows in each table across the 40 source codes was computed and the results can be found in the appendix.

Effectiveness of Optimisations

To test the effectiveness of our optimisations, the optimisations were toggled on and off and tested on two different sets of Autotester test cases, one being an adversarial set of test cases created to specifically target a lack of optimisation, and the other being a regular set of test cases which were used to test the correctness of Query Evaluator. More details on the test cases can be found in Section 4.4.2. For the correctness test case, the baseline used was the Query Evaluator without any optimisations. Each optimisation was then incrementally added, and the average time taken to evaluate each query was taken across 5 runs of the test case.

For the optimisation test case, as running the test case without any optimisations will result in very long TLEs, the baseline used was the Query Evaluator with all optimisations turned on, and each optimisation was toggled off individually to record any improvements or regression in total time taken.

More details on these timings can be found in the appendix. Note that the fifth optimization, sorting of neighbours, was not included in our final SPA due to its regressive effect.

3.5.5.5 Design Decisions for the Query Optimizer

Problem Statement: Method of sorting clauses or ResultTables such that intermediate size remains small. This saves some time in the inner join algorithm, especially if the intermediate tables need to be hashed.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Every join between tables should be an inner join
The sorting should be done such that cross joins are avoided, hence any two ResultTables that are merging must share at least one similar synonym.

Evaluation Criteria:

1. Efficiency of the query evaluation as a whole

Description of Possible Solutions:

Sort clauses before evaluation
Clauses can be sorted by the expected size of their ResultTable. Since the final order of ResultTables in the Database is maintained to the same as the order of evaluation of the clause, if the clauses are sorted in the correct order, ResultTables should be sorted from smallest to largest as well.
Sort Neighbours
During BFS, before the merging of each node, the neighbours of the node can be sorted such that tables are merged from smallest to largest.
Sort ResultTable before BFS
ResultTables can be sorted after the evaluation of all clauses, before any BFS is conducted.

Evaluation of solutions

Efficiency of the query evaluation	
Sort Clauses	The sorting of clauses in increasing order of expected table size not only helps with the merging but also the evaluation of clauses. If clauses with small table size (e.g Follows) are evaluated before larger relationships (e.g NextBip*), the larger relationship can be evaluated in the reduced space from the previous clause. However, since the result table size can only be estimated based on heuristics before evaluation of clauses, it might result in inaccurate sorting.
Sort Neighbours	This basically guarantees that neighbours are merged from smallest to largest. However, it does not guarantee that the BFS starts from the smallest possible table. If the first Clause evaluated happens to be the largest table, then the intermediate tables would contain the largest table from the very beginning. The additional cost of sorting neighbours is O(E log E) where E is the number of edges, where each edge represents the relationship between two clauses with a shared synonym.
Sort ResultTable before BFS	All ResultTables are sorted before BFS, hence it ensures that merging starts from the smallest possible table. However, the neighbours may not be in sorted order. However, since the first table is sufficiently small, the intermediate result space should remain relatively small.
Note: The efficiency of each query solution was also measured by the tests conducted as per Section 3.5.5.4.	

Final Choice:

Sort Clauses + Sort ResultTables Before BFS. Sorting of clauses helps to reduce the time taken in the Query Evaluation, and sorting result tables before BFS helps to correct any possible error in estimation of ResultTable size before merging.

3.5.6 Query Projector

The Query Projector is a small subcomponent of the Query Processing Subsystem. It is responsible for obtaining a QueryResult from the Query Evaluator, and formatting it into a list to return.

The exposed method for this class is the method `FormatResult`. It performs the following roles:

1. Identifies whether the output elements are integers, strings, boolean, or tuples.
2. Converts elements to strings if necessary.
3. Adds each element to the result list, which is then returned.

Projection of Example

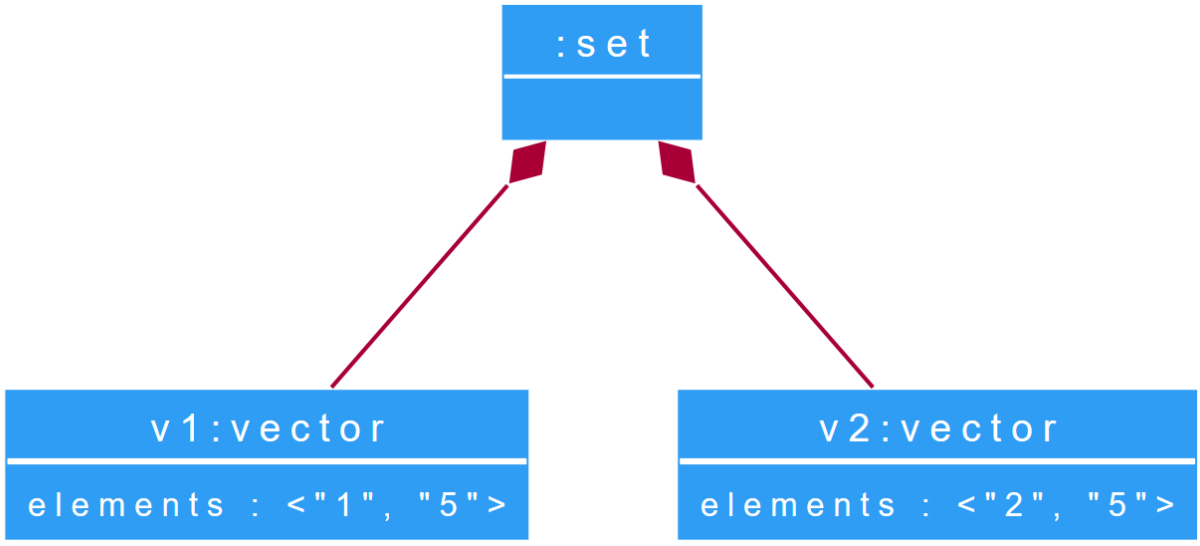

Figure 3.5.6.1: The state of the QueryResult object passed to the Query Projector

FIgure 3.5.6.1 shows the state of the QueryResult object passed as a parameter to the Query Projector. The role of the Query Projector is to extract the information from the QueryResult object and produce the formatted output.

For this QueryResult, the expected output is:

```
1 5, 2 5
```

This is the final output that will be returned from the Query Processor.

3.5.7 Design Decisions for Query Processor

Problem Statement: Choice of organization of Query Processor subcomponents.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Execution of core parsing and evaluation requirements
The Query Processor as a whole must be able to parse raw query strings, evaluate the contents, and output the result in a suitable format.

Evaluation Criteria:

1. Separation of Concerns
2. Memory Usage

Description of Possible Solutions:

Multiple static subcomponents
The Query Processor is dependent on multiple static classes (e.g. parser, evaluator, projector) that each have one entrypoint.
Multiple object subcomponents
The Query Processor creates multiple processor objects (e.g. main parser object, main PKB object, main evaluator object) that each have one entrypoint.

Evaluation of solutions

Separation of Concerns

Multiple static subcomponents	Separation of Concerns is achieved similarly with both of these solutions. Crucially, the main functions of the Query Processor (i.e. parsing, evaluating, projecting) are all delegated to a specific class which each expose only one main method. This ensures that the Query Processor itself has no knowledge of the workings of its constituent methods.
Multiple object subcomponents	

Memory Usage	
Multiple static subcomponents	Less memory usage - static objects are already initialized, and therefore do not take up any more memory even after repeated executions on different queries.
Multiple object subcomponents	More memory usage - objects must be created and destroyed each time a new query is evaluated. This results in wasted memory space and time.

Final Choice:

- Multiple static subcomponents were used to create the Query Processor. This is primarily due to the more efficient memory allocation of static classes.

4 Testing

Test Driven Development (TDD) was largely adhered to. Unlike a simpler CRUD application, this program has much more edge cases and requires extensive testing. Such testing cannot be done manually in an efficient manner whenever new code is added or existing code is refactored. As such, whenever bugs and issues are discovered, it will always first be added as a unit/integration test, such that future modifications to the code will always ensure that this bug is accounted for.

However, the main disadvantage of following such a paradigm is the heavy initial investment of time needed to come up with test cases. Again, our team has come to the conclusion that the benefits accrued over time will largely outweigh this initial investment needed. Since any refactor, no matter major or minor, or any additional code added may introduce regressions, having automated tests will end up giving the developers peace of mind knowing that what has worked before should still continue to work.

4.1 Testing Plan

Legend

The following acronyms indicate the component and the person(s) in charge of the testing.

SP - Source Processor: Aaron

DE - Design Extractor: Jia Da and Zi Ying

PKB - Program Knowledge Base: Hung

QP - Query Parser: Samuel

QE - Query Evaluator: Jolyn

QE-PKB, Query Evaluator - Program Knowledge Base Integration: Jolyn and Hung

DE-PKB, Design Extractor - Program Knowledge Base Integration: Jia Da and Zi Ying

Iteration	Week	Type	Rationale	Activity	Done by
1	2	Planning	Code implementation is very minimal. As such, there is not much implementation for tests. Planning was mostly done this week to prepare for the coming weeks.	-	All
	3	Unit testing	As the different components are being implemented with more features, unit tests can be written in complement to ensure correctness of each individual component.	Each component should write comprehensive unit tests for the features implemented in Week 2.	All
	4	Integration testing	Since a breadth-first iterative software development model is used, it is important that all the components are well integrated before adding on more new features.	Selection of design entities without any clauses via QE from PKB. Extraction or read and print information into PKB from DE, as well as Follows/* relationships	QE-PKB DE-PKB
	5	Integration testing		Queries with one clause (Follows/* and Parent/* clauses or pattern clause)	QE-PKB

				Population of Parent/Parent* clauses from DE to PKB	DE-PKB
		System testing	By the end of the first iteration, the system should be a complete and integrated system. The next step would be to evaluate the compliance of the system with the corresponding requirements of the Basic SPA.	Linking up of SPA class with the Autotester. Invalid source code test case for Autotester	SP
	6	System testing		Autotester Test Case 1 for source program with complex syntax and simple queries Autotester Test Case 2 to verify the correctness of queries with all design abstractions Autotester Test Case 3 to verify correctness of queries with complex queries	TC1 - Aaron and Jolyn TC2 - Samuel and Jolyn TC3 - Zi Ying, Jia Da and Hung
2	7	Unit testing	Iteration 2 involves incremental addition of features to each component. Unit testing is still necessary to ensure that these components work on their own with newly added features.	Tests should focus on the new features. SP: Multiple procedures DE, PKB: Calls, Calls*	All

		Integration testing	Since Iteration 2 involves changing of the individual components, integration testing is required to make sure that these changes do not break the existing system.	QE/QP: Selection of BOOLEAN and multiple clauses Any newly added features in QE and DE should also be added to the QE-PKB and DE-PKB integration tests.	
	8	System testing	After making sure that the system is integrated after changes from Iteration 2, system testing is done again to ensure that it is still compliant with requirements.	Autotester Test Case 4 for features added in Week 7 and some of Week 8: Pattern clauses with full expressions, Calls, Calls*, ModifiesP, UsesP, Selection of BOOLEAN	Aaron
	9	Functional tests; Positive & negative tests; Usability tests	At this stage it is a good time to start writing tests for error handling to prepare for final iteration.	Autotester Test Case 5 for queries with multiple clauses as well as selection of tuples and attributes Autotester Test Case 6: Complex source code to test the extraction of Calls, Calls*, ModifiesP and UsesP	TC5 - Jolyn TC6 - Jia Da, Zi Ying, Hung

3	10	Unit testing Integration testing	Each component should remain fully tested and integrated with the final addition of features.	Each component should test the correctness after adding Next/Next*/Affects/Affects*, especially correctness of CFG in DE.	All
	11	System testing	Ensure that the system is working after adding optimizations.	Autotester Test Case 7: Test effectiveness and correctness of query evaluation after adding each Query Optimization	Aaron and Jolyn
	12	Acceptance & Stress testing	Each query has a timeout limit of 5s - do stress testing to ensure that queries fall within the time constraint.	Stress Test Case: Queries that return extremely large tuple that requires a cross join Autotester Test Case 7: Add more queries that could possibility incur a timeout and aim to optimize them	Aaron Jolyn
		System testing	Ensure that the system fulfills all Advanced SPA requirements after the feature freeze.	Autotester Test Case 8: Invalid queries Autotester Test Case 9: Complex queries	Sam Jolyn

				Autotester Test Case 10: Invalid source Autotester Test Case 11: Hard to parse source	Aaron Aaron
	13	All tests should be complete by Week 13. Autotester Test Cases should be refined and checked for completeness.			

4.2 Unit Testing

4.2.1 Source Processor

As discussed in Section 3.1, the parsing of the SIMPLE source program is split up into two phases: lexing and parsing. The unit tests mirror this separation.

4.2.1.1 Source Program Lexer

Unit tests for the lexer focused on handling edge cases arising from arbitrary white spaces and using reserved SPA keywords as procedure names and variable identifiers. The output of the lexer is compared against the expected token string for correctness.

Each unit test will ascertain that the lexer is able to tokenize a specific part of the SIMPLE grammar, starting from an empty procedure. The test will then later encompass a full valid SIMPLE source code. Each unit test will also be duplicated twice, one which does not contain any extra whitespaces, and one where there exists arbitrary whitespaces.

In the following tables, each specific test for each type of statement is laid out:

A SIMPLE program with empty procedures
NOTE: the lexer does not care if procedures are empty or not and thus can still tokenize such a SIMPLE program **Types of test cases (assume each test is done twice: with and without arbitrary whitespaces):** • A single empty procedure • A single empty procedure with “procedure” as the procedure name • Multiple empty procedures

A SIMPLE program with non container statements

NOTE: the lexer does not care if call statements are made to a non-existent procedure and thus can still tokenize such a SIMPLE program

Types of test cases (assume each test is done twice: with and without arbitrary whitespaces):

- A single procedure with read statement
- A single procedure with print statement
- A single procedure with call statement
- A single procedure with read/print/call statements
- A single procedure with read/print/call statements using reserved keywords like read/print/call as variable names
- A single procedure with assign statements
- A single procedure with assign statements using reserved keywords like read/print/call as variable names

Example of Test Case

Purpose: To test that the Lexer is able to correctly tokenize programs with read, print and call statements.

Required Input: A source program string as follows - `procedure main { read x; print y; call z; }`

Expected Output: A TokenList containing all the tokens used in the source program.

```
GIVEN( desc: "A SIMPLE program with 1 procedure and read/print/call statements") {
  TokenList correct_tokens;
  correct_tokens
      .Push(Token("procedure", TokenType::Procedure))
      .Push(Token("main", TokenType::ProcedureName))
      .Push(Token("{", TokenType::Parenthesis))
      .Push(Token("read", TokenType::Read))
      .Push(Token("x", TokenType::VariableName))
      .Push(Token(";", TokenType::Semicolon))
      .Push(Token("print", TokenType::Print))
      .Push(Token("y", TokenType::VariableName))
      .Push(Token(";", TokenType::Semicolon))
      .Push(Token("call", TokenType::Call))
      .Push(Token("z", TokenType::ProcedureName))
      .Push(Token(";", TokenType::Semicolon))
      .Push(Token("}", TokenType::Parenthesis));

  WHEN( desc: "There are no whitespaces") {
    std::string program( s: "procedure main{read x;print y;call z;}");
    TokenList lexer_tokens = Lexer::Tokenise(program);
    REQUIRE(correct_tokens == lexer_tokens);
  }

  WHEN( desc: "There are arbitrary whitespaces") {
    std::string program( s: "\n\n procedure\n main{ \nread  \n\n\n x  \n; \n\nprint y   ;\n\ncall  z ;\n } ");
    TokenList lexer_tokens = Lexer::Tokenise(program);
    REQUIRE(correct_tokens == lexer_tokens);
  }
}
```

A SIMPLE program with container statements
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): • A single procedure with a single if statement • A single procedure with a single while statement • A single procedure with nested if and while statements
Example of Test Case **Purpose:** To test that the Lexer is able to correctly tokenize nested if and while statements while correctly differentiating conditional expressions from normal

expressions.

Required Input: A source program string as follows:

```
procedure main {
  if (a <= b) then {
    a = a + 1;
    while (a != b) {
      b = b / 2;
      if (a == b) then {
        a = b;
      } else {
        c=d;
      }
    }
  } else {
    if (b > c) then {
      z = 2;
    } else {
      z = 4;
    }
  }
}
```

Expected Output: A TokenList containing all the tokens used in the source program.

```
GIVEN( desc: "A valid simple program with nested while and if statements") {
  TokenList correct_tokens;
  correct_tokens
      .Push(Token("procedure", TokenType::Procedure))
      .Push(Token("main", TokenType::ProcedureName))
      .Push(Token("{", TokenType::Parenthesis))
      .Push(Token("if", TokenType::If))
      .Push(Token("(", TokenType::Parenthesis))
      .Push(Token("a", TokenType::VariableName))
      .Push(Token("<=", TokenType::RelativeExpressionOp))
      .Push(Token("b", TokenType::VariableName))
      .Push(Token(")", TokenType::Parenthesis))
      .Push(Token("then", TokenType::Then))
      .Push(Token("{", TokenType::Parenthesis))
      .Push(Token("a", TokenType::VariableName))
      .Push(Token("=", TokenType::AssignmentOp))
      .Push(Token("a", TokenType::VariableName))
      .Push(Token("+", TokenType::ExpressionOp))
      .Push(Token("1", TokenType::ConstantValue))
      .Push(Token(";", TokenType::Semicolon))
      .Push(Token("while", TokenType::While))
      .Push(Token("(", TokenType::Parenthesis))
      .Push(Token("a", TokenType::VariableName))
      .Push(Token("!=", TokenType::RelativeExpressionOp))
      .Push(Token("b", TokenType::VariableName))
      .Push(Token(")", TokenType::Parenthesis))
      .Push(Token("{", TokenType::Parenthesis))
      .Push(Token("b", TokenType::VariableName))
      .Push(Token("=", TokenType::AssignmentOp))
      .Push(Token("b", TokenType::VariableName))
      .Push(Token("/", TokenType::ExpressionOp))
      .Push(Token("2", TokenType::ConstantValue))
      .Push(Token(";", TokenType::Semicolon))
      .Push(Token("if", TokenType::If))
      .Push(Token("(", TokenType::Parenthesis))
      .Push(Token("a", TokenType::VariableName))
      .Push(Token("==", TokenType::RelativeExpressionOp))
      .Push(Token("b", TokenType::VariableName))
      .Push(Token(")", TokenType::Parenthesis))
```

```
    .Push(Token("a", TokenType::VariableName))
    .Push(Token("==", TokenType::RelativeExpressionOp))
    .Push(Token("b", TokenType::VariableName))
    .Push(Token(")", TokenType::Parenthesis))
    .Push(Token("then", TokenType::Then))
    .Push(Token("{", TokenType::Parenthesis))
    .Push(Token("a", TokenType::VariableName))
    .Push(Token("=", TokenType::AssignmentOp))
    .Push(Token("b", TokenType::VariableName))
    .Push(Token(";", TokenType::Semicolon))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("else", TokenType::Else))
    .Push(Token("{", TokenType::Parenthesis))
    .Push(Token("c", TokenType::VariableName))
    .Push(Token("=", TokenType::AssignmentOp))
    .Push(Token("d", TokenType::VariableName))
    .Push(Token(";", TokenType::Semicolon))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("else", TokenType::Else))
    .Push(Token("{", TokenType::Parenthesis))
    .Push(Token("if", TokenType::If))
    .Push(Token("(", TokenType::Parenthesis))
    .Push(Token("b", TokenType::VariableName))
    .Push(Token(">", TokenType::RelativeExpressionOp))
    .Push(Token("c", TokenType::VariableName))
    .Push(Token(")", TokenType::Parenthesis))
    .Push(Token("then", TokenType::Then))
    .Push(Token("{", TokenType::Parenthesis))
    .Push(Token("z", TokenType::VariableName))
    .Push(Token("=", TokenType::AssignmentOp))
    .Push(Token("2", TokenType::ConstantValue))
    .Push(Token(";", TokenType::Semicolon))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("else", TokenType::Else))
    .Push(Token("{", TokenType::Parenthesis))
    .Push(Token("z", TokenType::VariableName))
```

```
    .Push(Token("=", TokenType::AssignmentOp))
    .Push(Token("4", TokenType::ConstantValue))
    .Push(Token(";", TokenType::Semicolon))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("}", TokenType::Parenthesis))
    .Push(Token("}", TokenType::Parenthesis));

  WHEN( desc: "There are no whitespaces") {
    std::string program( s: "procedure main{if(a<=b)then{a=a+1;while(a!=b){b=b/2;if(a==b)"
                                "then{a=b;}else{c=d;}}}else{if(b>c)then{z=2;}else{z=4;}}}");
    TokenList lexer_tokens = Lexer::Tokenise(program);
    REQUIRE(correct_tokens == lexer_tokens);
  }

  WHEN( desc: "There are arbitrary whitespaces") {
    std::string program( s: "procedure main  \n{ \n if  \n(a \n<= \nb ) then {\n\n a= \na + 1 ; while \n(a\n !=  b ) "
                        "\n{\nb = \nb/  2 ; \nif \n(a == b) \n\nthen \n{ \n\na=b \n;} \nelse{c\n=d \n;} \n} \n} \nelse"
                        "\n { \nif( \nb \n>  c ) \nthen \n{ \nz= \n2; } \nelse{z= 4\n\n;}\n} \n}\n  ");
    TokenList lexer_tokens = Lexer::Tokenise(program);
    REQUIRE(correct_tokens == lexer_tokens);
  }
}
```

4.2.1.2 Source Program Parser

Unit tests for the parser focused on correctness of the AST produced, and if the parser is able to reliably detect invalid SIMPLE source programs. For a valid test, the output of the parser is compared against the expected hardcoded AST for correctness.

In the following tables, each valid test is laid out:

A valid SIMPLE program with non-container statements
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): • A single valid procedure with read/print statement • Multiple valid procedures with read/print statement • Multiple valid procedures with call statements • A single valid procedure with a simple assign statement • A single valid procedure with a complex assign statement

- Duplicate tests for procedure(s) with use of reserved keywords as variable/procedure names

Example of Test Cases

Purpose: To test that the Parser is able to parse a SIMPLE program with assign statements, with and without arbitrary whitespaces

Required Input: A source code string for the program

Expected Output: An AST containing TNodes that correspond to the program. A handcrafted AST is used to check against the expected output.

```
GIVEN( desc: "A valid SIMPLE program with 1 simple assign statement") {
  TNode correct_ast(TNodeType::Program);
  TNode proc_node(TNodeType::Procedure, std::string( s: "main"));
  TNode stmt_list_node(TNodeType::StatementList);
  TNode assign_node(TNodeType::Assign, 1);
  TNode var_node(TNodeType::Variable, 1, "myGoodVar420");
  TNode const_node(TNodeType::Constant, 1, "69");
  correct_ast.AddChild(&proc_node.AddChild(&stmt_list_node.AddChild(
      &assign_node.AddChild(&var_node).AddChild(&const_node))));

  WHEN( desc: "There are no whitespaces") {
    std::string program( s: "procedure main{myGoodVar420=69;}");
    TNode parser_ast = Parser::Parse(program);
    REQUIRE(IsSimilarAST(correct_ast, parser_ast));
  }

  WHEN( desc: "There are arbitrary whitespaces") {
    std::string program( s: "procedure main \n\n  \n{  \nmyGoodVar420=  \n\n69  \n;  \n}");
    TNode parser_ast = Parser::Parse(program);
    REQUIRE(IsSimilarAST(correct_ast, parser_ast));
  }
}
```

A valid SIMPLE program with container statements

Types of test cases (assume each test is done twice: with and without arbitrary whitespaces):

- A single valid procedure with a while statement
- A single valid procedure with an if statement
- A complex procedure with nested container statements and all non-container statements with use of reserved keywords as variable/procedure names

In the following tables, each invalid test is laid out:

An invalid SIMPLE program with empty statement lists
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): ● A single empty procedure ● A procedure with an empty while statement ● A procedure with an empty if statement ● A procedure with a nested empty while statement ● A procedure with a nested empty if statement ● Multiple procedures with exactly one empty procedure
Example of Test Cases **Purpose:** To test that the Parser is able to identify when any statement lists within a SIMPLE program is empty. **Required Input:** A source code string for the program **Expected Output:** Error should be thrown and the user notified accordingly

```
GIVEN( desc: "An invalid SIMPLE program with empty statement lists") {
  WHEN( desc: "There are no whitespaces") {
    REQUIRE_THROWS(Parser::Parse("procedure main{}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{while(1==1){}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{if(1==1)then{}else{}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{if(1==1)then{x=1;}else{}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{if(1==1)then{}else{x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{if(1==1)then{while(1==1){}}else{x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{if(1==1)then{x=1;}else{while(1==1){}}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{read x;}procedure bar{}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{}procedure bar{read x}"));
  }

  WHEN( desc: "There are arbitrary whitespaces") {
    REQUIRE_THROWS(Parser::Parse(" \n procedure \n\n main  \n  {  \n   \n  }  \n  \r\t  "));
  }
}
```

An invalid SIMPLE program with multiple procedures with the same name
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): • Two procedures of the same name • Three procedures with all of the same name • Three procedures with exactly two of the same name

An invalid SIMPLE program with invalid call statements
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): • Two procedures with one calling a non-existent procedure • Three procedures with one calling a non-existent procedure • One procedure with self recursive call statement

- Two procedures with recursive call statements
- Multiple procedures which eventually has a recursive call statement

Example of Test Case

Purpose: To test that the Parser can detect invalid call statements and notify the users appropriately.

Required Input: Program strings that include all of the above cases.

Expected Output: Throws error when invalid call statements are detected.

```
GIVEN( desc: "An invalid SIMPLE program calling non existent procedures") {
  REQUIRE_NOTHROW(Parser::Parse("procedure foo{read x;}procedure bar{call foo;}"));                      // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure foo{call koo;}procedure bar{call foo;}procedure koo{read y;}"));  // this is ok
  WHEN( desc: "There are no whitespaces") {
    REQUIRE_THROWS(Parser::Parse("procedure foo{read x;}procedure bar{call koo;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call goo;}procedure bar{read x;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call goo;}procedure bar{call koo;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call goo;}procedure bar{call koo;}procedure koo{read y;}"));
  }
}

GIVEN( desc: "An invalid SIMPLE program with recursive procedure calls") {
  REQUIRE_NOTHROW(Parser::Parse("procedure foo{read x;}procedure bar{call foo;}"));                      // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure foo{call koo;}procedure bar{call foo;}procedure koo{read y;}"));   // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure main{call koo;}procedure bar{call koo;}procedure koo{read y;}"));  // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure main{call a;call b;call c;}procedure a{call b;call d;}procedure b{call c;}procedure c{read y;}procedure d{call b;}"));
  WHEN( desc: "There are no whitespaces") {
    REQUIRE_THROWS(Parser::Parse("procedure foo{read x;}procedure bar{call bar;}"));  // self recursive
    REQUIRE_THROWS(Parser::Parse("procedure foo{call bar;}procedure bar{call foo;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call bar;}procedure bar{call koo;}procedure koo{call foo;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call bar;}procedure bar{call koo;}procedure koo{call bar;}"));
    REQUIRE_THROWS(Parser::Parse("procedure foo{call koo;}procedure bar{read x;}procedure koo{call foo;}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{call a;call b;call c;}procedure a{call b;call d;}procedure b{call c;}procedure c{read y;}procedure d{call a;}"));
  }
}
```

An invalid SIMPLE program with invalid procedure syntax

Types of test cases (assume each test is done twice: with and without arbitrary whitespaces):

- Procedures without names
- Misspelt procedure keyword (ie. “procdure”)
- Procedures with missing/mismatched curly braces
- Procedures with names starting with a number
- Procedures without statement lists

An invalid SIMPLE program with invalid assign statement syntax
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): ● Assign statements ending with more than a single semicolon ● Missing RHS ● Missing LHS ● LHS with two variables ● RHS containing illegal tokens like “>”

An invalid SIMPLE program with invalid if/while statement syntax
Types of test cases (assume each test is done twice: with and without arbitrary whitespaces): ● Missing statement list ● Invalid conditional expression ● Invalid arbitrary parentheses surrounding a relative expression ● Invalid arbitrary parentheses surrounding a conditional expression
Example of Test Case **Purpose:** To test that the Parser accepts arbitrary parenthesis within a relative expression but correctly throws error when there are arbitrary parenthesis around a conditional expression. **Required Input:** Program strings that include both arbitrary parenthesis within and outside a relative expression. **Expected Output:** Throws error when arbitrary parentheses is detected in a conditional expression.

```
GIVEN( desc: "Invalid arbitrary parentheses surrounding a relative expression") {
  // arbitrary parentheses is ok in expressions within a relative expression
  REQUIRE_NOTHROW(Parser::Parse("procedure main{while(a==b){x=1;}}"));          // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure main{while(a==(b)){x=1;}}"));        // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure main{while((a)==(b)){x=1;}}"));      // this is ok
  REQUIRE_NOTHROW(Parser::Parse("procedure main{while(((a))==((b))){x=1;}}"));  // this is ok
  WHEN( desc: "There are no whitespaces") {
    REQUIRE_THROWS(Parser::Parse("procedure main{while((a==b)){x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{while(((a==b))){x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{while((a==)b){x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{while(a(==)b){x=1;}}"));
    REQUIRE_THROWS(Parser::Parse("procedure main{while(a(==b)){x=1;}}"));
  }
}
```

4.2.2 Design Extractor

For iteration 3, the decision to not do unit testing of the DesignExtractor was made, as it was too costly, time- and effort-wise, for the benefit it gave. Since the DesignExtractor has soft dependencies on the Source Parser and PKB, unit tests for the DesignExtractor require handcrafting one AST and PKBStub *per* test. Manual construction of a non-trivial AST that would meaningfully test the DesignExtractor easily reached a few hundred AST nodes, and hence was not feasible to do. Further, each PKBStub had to be made specifically for the input AST. These 2 large costs meant that the DE could not be feasibly unit tested with many tests, or non-trivial tests, defeating the purpose of unit testing.

However, since the Source Parser and PKB were extensively unit tested, we had confidence to proceed with integration tests for the DesignExtractor using the actual Source Parser and PKB. Without the need to manually construct the AST and PKBStub for every test, we could write a greater number of complex SIMPLE sources which properly stressed the DesignExtractor.

4.2.3 PKB

The PKB conducted unit testing for its template classes, each design entity and design abstraction table as well as the overall PKB's APIs. For each template class, different hash tables containing mapping of different data types were instantiated for testing of all common table operations. For instance, the followings are testing scenarios for `TableSingle` template class:

	`TableSingle` test cases:
Instantiation	**NOTE: The following test cases are all invalid.** **Types of test cases:** ● `TableSingle<int, string>` ● `TableSingle<string, int>` ● `TableSingle<int, int>` ● `TableSingle<string, string>` ● `TableSingle<int, TokenList> for AssignTable` ● `TableSingle<string, pair<int, int>> for ProcTable`
Insertion	**NOTE: The following test cases are all invalid which would return false.** **Types of test cases:** ● Insert with an existing key and the same value ● Insert with an existing key and a different value ● Insert with a new valid key but invalid value ● Insert with an invalid key but valid value ● Insert with an invalid key and invalid value
Get	**Types of test cases:**

	• `Get` called with a valid key
	• `Get` called with an invalid key
	  ○ `TableSingle<int, string>` returns empty string
	  ○ `TableSingle<string, int>` returns -1
	  ○ `TableSingle<int, int>` returns -1
	  ○ `TableSingle<string, string>` returns empty string
	  ○ `TableSingle<int, TokenList>` returns empty `TokenList`
	  ○ `TableSingle<string, pair<int, int>>` returns empty `pair`

Contains	**Types of test cases:** • `Contains` called with an existing key returns true • `Contains` called with a non-existing key returns false
GetAllKeys	**Types of test cases:** • `TableSingle<int, string>` returns correct set of integers • `TableSingle<string, int>` returns correct set of strings • `TableSingle<int, int>` returns correct set of integers • `TableSingle<string, string>` returns correct set of strings • `TableSingle<int, TokenList>` returns correct set of integers • `TableSingle<string, pair<int, int>>` returns correct set of strings
Example of Test Case **Purpose:** To test that the `TableSingle` template class can instantiate different hash table objects which can operate with different key and value data types. **Required Input:** There is no required input. **Expected Output:** Different hash table objects with the corresponding key and value	

data types are created and currently empty.

```
GIVEN( desc: "Different template class TableSingle construction.") {
  TableSingle<int, std::string> table_int_string;
  TableSingle<std::string, int> table_string_int;
  TableSingle<int, int> table_int_int;
  TableSingle<int, source_processor::TokenList> table_int_tokenlist;
  TableSingle<std::string, std::pair<int, int>> table_string_pair;
  THEN( desc: "TableSingle exists with size 0.") {
    REQUIRE(table_int_string.IsEmpty());
    REQUIRE(table_string_int.IsEmpty());
    REQUIRE(table_int_int.IsEmpty());
    REQUIRE(table_int_tokenlist.IsEmpty());
    REQUIRE(table_string_pair.IsEmpty());
  }
}
```

After the template classes, each design entity and design abstraction tables were also tested. The test cases were designed to test valid and invalid insertions into the tables given different existing conditions, then validated to see if the boolean value for each insertion was correctly returned and checked if all valid insertions were correctly stored in their respective tables. At the same time, it also checked whether the inverse tables were populated with each insertion. For design abstractions, relationship APIs were additionally checked. Afterwards, retrieval APIs were also tested to ensure the correctness of the results returned. The following table illustrates how a relationship table such as `ParentTable` is unit tested:

	`ParentTable` test cases:
Instantiation	**Template Tables for storage:** • `TableMultiple<int, int> parent_table` • `TableSingle<int, int> inverse_parent_table` **Types of test cases:**

	• `Parent_table` initialized to empty • `Inverse_parent_table` initialized to empty
Insertion	**Types of test cases:** • Valid `InsertParent(stmt1, stmt2)` returns true • Invalid insertion with `stmt1 <= 0 and stmt1 >= stmt2` returns false • Invalid insertion with `stmt2` already exists in inverse table (`stmt2` can only have one parent) returns false
Relationship checking	**Types of test cases:** • `IsParent(stmt1, stmt2)` returns true if `stmt2` is a value within the set of values at `stmt1` and false otherwise • `IsParent(stmt1, stmt2)` with invalid index `stmt1 <= 0 and stmt1 >= stmt2` returns false
Retrieval with parameter	**Types of test cases:** • `GetChildrenStatements(stmt1)` returns correct set of statements at `stmt1` • `GetChildrenStatements(stmt1)` returns an empty set if `stmt1` is not an existing key in `parent_table` • `GetParentStatement(stmt2)` returns correct parent statement `stmt1` • `GetParentStatement(stmt2)` returns -1 if `stmt2` is not an existing key in `inverse_parent_table`
Retrieval all	**Types of test cases:** • `GetAllParentStmts` returns correct set of all `Parent` statements stored in `parent_table` • `GetAllChildrenStmts` returns correct set of children statements stored in `inverse_parent_table`

Example of Test Case

Purpose: To test that the `ParentTable` successfully inserts and stores valid `Parent(stmt1, stmt2)` relationships and populates the inverse table as well.

Required Input: Valid `Parent(stmt1, stmt2)` relationships

Expected Output: Both the direct and inverse template tables for `ParentTable` are correctly populated and return True upon a successful insertion.

```
WHEN( desc: "Insert a valid Parent(stmt1, stmt2) relationship.") {
  THEN( desc: "Insertion returns True. Both parent_table and inverse_parent_table are filled.") {
    REQUIRE(parent_table.InsertParent(5, 7));
    REQUIRE(parent_table.InsertParent(5, 8));
    REQUIRE(parent_table.InsertParent(10, 11));
    REQUIRE(parent_table.InsertParent(10, 13));
    TableMultiple<int, int> parent = parent_table.GetParentTable();
    TableSingle<int, int> inverse_parent = parent_table.GetInverseParentTable();
    REQUIRE(parent.Size() == 2);
    REQUIRE(parent.Get(5).size() == 2);
    REQUIRE(ContainsExactly(parent.Get(5), {7, 8}));
    REQUIRE(parent.Get(10).size() == 2);
    REQUIRE(ContainsExactly(parent.Get(10), {11, 13}));
    REQUIRE(inverse_parent.Size() == 4);
    REQUIRE(ContainsExactly(inverse_parent.GetAllKeys(), {7,8,11,13}));
  }
```

For the PKB's unit testing, all APIs were tested once again to ensure the PKB is a good encapsulation of all its underlying entity and relationship tables. Since the PKB also utilizes its tables' respective APIs to perform most operations, the unit tests will not be illustrated again here for conciseness.

4.2.4 Query Processor

The Query Processor conducted unit testing using a PKBStub, which was modelled after Code 3: Compute the sum of the digits of an integer in SIMPLE provided in the Basic SPA Requirements. The PKBStub directly inherits the PKB and overrides functions such as `IsFollows(s1, s2)` to simplistically return when s1 and s2 is equivalent to certain predetermined values.

4.2.4.1 Query Parser

The testing performed on the Query Parser consists mainly of Unit Testing. This is because it is difficult for the Query Parser to drive integration tests (since it is entirely contained within itself).

Each unit test seeks to test one specific rule of the PQL concrete grammar. A valid test case would seek to test some combination of expected valid inputs, and an invalid test case would test one invalid part of an input, with the rest of the input being valid.

The proposed unit testing plan for the Query Parser is shown below. The specific features tested by the test cases are bolded. Each test case involves calling the main method `GenerateQuery`, and comparing the generated result with the expected result.

Declaration clauses are syntactically valid
NOTE: The following test cases are all invalid, i.e. they should all throw an error. **Types of test cases:** ● Clause does not contain **design entity type** ● Clause does not contain **at least one synonym** ● Clause contains terms that are **not valid synonyms** ● Clause contains multiple synonyms **without comma** separating them ● Clause does not **end in semicolon** ● Clause comes after ‘Select’ clause
Example of test cases: **Purpose:** To test that an invalid synonym in declaration clauses is correctly identified. **Required Input:** A raw query string. **Expected Output:** A runtime error thrown by the main method of the Query Parser.

```
WHEN( desc: "The query's declaration has an invalid synonym name.") {
    string INVALID_DECLARATION_SYNONYM_STRING = "stmt _s; Select _s";

    THEN( desc: "GenerateQuery() throws an std::runtime_error.") {
        REQUIRE_THROWS(QueryParser::GenerateQuery(INVALID_DECLARATION_SYNONYM_STRING));
    }
}
```

Declaration clauses are semantically valid
NOTE: The following test cases are all invalid, i.e. they should all throw an error. **Types of test cases:** • **Synonym is re-declared** in separate declaration clause

Selected Entity is syntactically valid
NOTE: The following test cases are all invalid, i.e. they should all throw an error. **Types of test cases:** • Selected Entity is of **wrong type** (i.e. not BOOLEAN, synonym, attribute, or tuple) • Selected Entity is **tuple with wrong type value** (i.e. not synonym or attribute)

Selected Entity is semantically valid

NOTE: The following test cases are all invalid, i.e. they should all throw an error.

Types of test cases:

- Selected Entity is **undeclared synonym**
- Selected Entity is **attribute with undeclared synonym**
- Selected Entity is attribute with **incorrect attribute type**
- Selected Entity is attribute with **non-matching** synonym type and attribute type
- Selected Entity is **tuple** with one element out of many being undeclared

Selected Entity is parsed correctly

NOTE: The following test cases involve queries with no conditional clauses. In each test case, the generated output is compared to the expected Query.

Types of test cases:

- **BOOLEAN** is selected
- **Synonym** is selected
- **Attribute** is selected
- Tuple with **single element** is selected
- Tuple with **multiple elements** of same type is selected
- Tuple with multiple elements of **different type** is selected
- Tuple with **duplicate elements** is selected
- Tuple with elements that share **same synonym but different attributes** is selected
- Tuple with **synonyms named after keywords** is selected
- Tuple with **excessive valid whitespace** is selected

Such That clauses are syntactically valid
NOTE: The following test cases are all invalid, i.e. they should all throw an error. Additionally, this logic is used for Pattern and With clauses (which are omitted for the sake of brevity). **Types of test cases:** ● **Missing key tokens** (e.g. parentheses, commas) ● **Incorrect design abstraction** used ● **Incorrect parameter type** of design abstraction used ● **Invalid space** between design abstraction and asterisk (e.g. Follows *)

Such That clauses are semantically valid
NOTE: The following test cases are all invalid, i.e. they should all throw an error. Additionally, this logic is used for Pattern and With clauses (which are omitted for the sake of brevity). **Types of test cases:** ● Parameter contains **undeclared synonym**

Such That clauses are parsed correctly
NOTE: This logic is used for Pattern and With clauses (which are omitted for the sake of brevity). In each test case, the generated output is compared to the expected Query.

Types of test cases:

- **Valid combination of parameters** (i.e. integer/synonym/string/wildcard) used in single clause
- **Synonyms named after keywords** are used in single clause
- **Multiple clauses** of same type are used
- Clauses contain **excessive valid whitespace**

Example of test cases:

Purpose: To test that the Query Parser can handle adversarially-named synonyms in the query string.

Required Input: A raw query string with synonym names that mirror PQL keywords, and a constructed Query object with the expected correct structure.

Expected Output: A Query object that is equal to the constructed Query object.

```
WHEN("Modifies clause uses synonyms with design entity type names.") {
  string CONFUSING_SYNONYM_NAMES_STRING = "stmt stmt, Modifies; while while, if, Select; Select Select such that Modifies(if, Select)";
  Query GENERATED_QUERY = QueryParser::GenerateQuery(CONFUSING_SYNONYM_NAMES_STRING);

  Query EXPECTED_QUERY = Query(SelectedEntity(DesignEntity(DesignEntityType::WHILE, "Select")));
  EXPECTED_QUERY.AddClause(SuchThatClause(DesignAbstraction::MODIFIES,
                                          ClauseParam(DesignEntity(DesignEntityType::WHILE, "if")),
                                          ClauseParam(DesignEntity(DesignEntityType::WHILE, "Select"))));

  THEN("GenerateQuery() returns the expected Query.") {
    REQUIRE(GENERATED_QUERY.GetSelectedEntity().entity_type == SelectedEntityType::DESIGN_ENTITY);
    REQUIRE(GENERATED_QUERY.GetSelectedEntity().design_entity == EXPECTED_QUERY.GetSelectedEntity().design_entity);
    REQUIRE(GENERATED_QUERY.GetClauseList().size() == EXPECTED_QUERY.GetClauseList().size());
    REQUIRE(GENERATED_QUERY.GetClauseList().at(0) == EXPECTED_QUERY.GetClauseList().at(0));
  }
}
```

Multiple clauses are parsed correctly

NOTE: In each test case, the generated output is compared to the expected Query.

Types of test cases:

- Multiple clauses of **different types** are used without ‘and’

- Multiple clauses of same types are used with **‘and’ directly connecting them**
- Multiple clauses of same types are used with ‘and **indirectly** connecting them
- Multiple clauses of **different types** are used, with the same types **indirectly** connected with ‘and’

Example of test cases:

Purpose: To test that the Query Parser can handle ‘and’ keywords that are used to directly connect 2 clauses of the same type.

Required Input: A raw query string with two clauses of the same type connected with ‘and’, and a constructed Query object with the expected correct structure.

Expected Output: A Query object that is equal to the constructed Query object.

```
WHEN( desc: "The query uses 'and' to directly connect two clauses.") {
string MULTIPLE_PATTERN_CLAUSE_STRING = "stmt s; assign a, a1; variable v; Select a pattern a(\"string\", _\"a+b\"_) and a1(v, \"b+c/d\")";
Query GENERATED_QUERY = QueryParser::GenerateQuery( &: MULTIPLE_PATTERN_CLAUSE_STRING);

Query EXPECTED_QUERY = Query(SelectedEntity( de: DesignEntity(DesignEntityType::ASSIGN, "a")));
EXPECTED_QUERY.AddClause(PatternClause(
    DesignEntity(DesignEntityType::ASSIGN, "a"),
    ClauseParam( var_proc_name: "string"),
    ClauseParam( pattern_expr: PatternExpression( token_list: parser_utils::ExpressionParser::ParseExpression("a+b"), is_wild_card: true))));
EXPECTED_QUERY.AddClause(PatternClause(
    DesignEntity(DesignEntityType::ASSIGN, "a1"),
    ClauseParam( design_entity: DesignEntity(DesignEntityType::VARIABLE, "v")),
    ClauseParam( pattern_expr: PatternExpression( token_list: parser_utils::ExpressionParser::ParseExpression("b+c/d"), is_wild_card: false))));

THEN( desc: "GenerateQuery() returns the expected Query.") {
  REQUIRE(GENERATED_QUERY.GetSelectedEntity().entity_type == SelectedEntityType::DESIGN_ENTITY);
  REQUIRE(GENERATED_QUERY.GetSelectedEntity().design_entity == EXPECTED_QUERY.GetSelectedEntity().design_entity);
  REQUIRE(GENERATED_QUERY.GetClauseList().size() == EXPECTED_QUERY.GetClauseList().size());
  REQUIRE(GENERATED_QUERY.GetClauseList().at(0) == EXPECTED_QUERY.GetClauseList().at(0));
  REQUIRE(GENERATED_QUERY.GetClauseList().at(1) == EXPECTED_QUERY.GetClauseList().at(1));
 }
}
```

4.2.4.2 Query Evaluator

The Query Evaluator test cases are generally divided by single clause and multiple clause queries. There are also some test cases that are more specific to the selection of the query. In the following test cases, INT refers to statement indexes, NAME refers to strings that might be variable names or procedure names, and DE refers to some synonym that represents a design entity (e.g s for stmt, pn for print).

Unit tests in the Query Evaluator are conducted on the PKBStub, which is a class that inherits from the PKB and implements methods called by the Query Evaluator. The

methods are implemented with no logic involved, but based on a particular piece of source code, e.g `IsFollows(int s1, int s2)` will only return true if `s1 == 1 && s2 == 2` based on the source code.

Single clause tests
Note: In the following clause, SuchThatClause refers to any of the following design abstractions: Follows, Follows*, Parent, Parent*, Modifies, Uses, Calls, Calls* unless otherwise specified ● SuchThatClause(DE, DE) where the two synonyms are **exactly identical** ● SuchThatClause(DE, DE) where the two synonyms are of the **same design entity type** but different synonym name ● SuchThatClause(**INT**, DE) ● SuchThatClause(**NAME**, DE) ● SuchThatClause(**DE**, _) ● SuchThatClause(_, _) ● pattern **a(DE, pattern expr)** ● pattern **while(DE, _)** ● pattern **ifs(DE, _, _)** ● with INT/NAME = with INT/NAME ● with ATTR = ATTR where the two attributes are of different types return false ● with ATTR = ATTR where the attributes returns the same result as the design entity (e.g `pn` and `pn.stmt#`) ● with ATTR = ATTR where the attributes return a type different from the native design entity (e.g `pn.varName`)

Note that while some single clause tests were conducted in unit testing, these tests are mostly related to implementation of logic details in the Query Evaluator. Extensive single clause tests for each design abstraction are conducted in the integration tests to also

ensure the integrity of each design abstraction table in the PKB. This will be further elaborated in Section 4.2.2.

Multiple clause tests
For each of the following test cases, generally there is at least one test that returns true and one test that returns false unless the query specifically returns true or false. For each test heuristic, it is generally applied multiple times to cover testing of all design abstractions and types of clauses. ● Multiple clauses **evaluate independently to true without evaluating any DEs** (i.e all INTs or NAMEs) ● Multiple clauses **evaluate independently to true with DEs** (i.e none of the clauses have any overlapping parameters) ● Multiple clauses evaluate independently, **one or a few to false** ● Multiple clauses with **overlapping synonyms**, later clauses change the result of previous parameters (e.g `Follows(s1, s2) and Modifies(s2, v)`, evaluation of `Modifies(s2, v)` results in a change in `s1` even though it was not evaluated) ● Multiple clauses with overlapping synonyms, all clauses evaluate to true independently but the **intersection of the clauses is empty** (hence the query results to false) ● Multiple clauses with **completely identical parameters** (e.g `Uses(a,v) and Modifies(a,v)`)
Example of test cases **Purpose:** To test the ability of the Query Evaluator to handle clauses that share overlapping synonyms **Required Input:** A Query object with 5 clauses that each share at least one overlapping synonym with another clause **Expected Output:** The assignment statement that uses and modifies the same

variable, and is followed directly or indirectly by a statement that modifies a variable that is also modified by some read statement {stmt 5 of the source code from PKBStub}

```
WHEN( desc: "Three or more SuchThatClauses and PatternClauses evaluate the overlapping synonym pairs return true") {
  Query valid_query = Query(SelectedEntity( de: DesignEntity(DesignEntityType::ASSIGN, "a")));
  SuchThatClause uses_clause = SuchThatClause(DesignAbstraction::USES,
                                              ClauseParam( design_entity: DesignEntity(DesignEntityType::ASSIGN, "a")),
                                              ClauseParam( design_entity: DesignEntity(DesignEntityType::VARIABLE, "v")));
  SuchThatClause follows_clause = SuchThatClause(DesignAbstraction::FOLLOWS_T,
                                                 ClauseParam( design_entity: DesignEntity(DesignEntityType::ASSIGN, "a")),
                                                 ClauseParam( design_entity: DesignEntity(DesignEntityType::STMT, "s")));
  SuchThatClause modifies_clause1 = SuchThatClause(DesignAbstraction::MODIFIES,
                                                   ClauseParam( design_entity: DesignEntity(DesignEntityType::STMT, "s")),
                                                   ClauseParam( design_entity: DesignEntity(DesignEntityType::VARIABLE, "v1")));
  SuchThatClause modifies_clause2 = SuchThatClause(DesignAbstraction::MODIFIES,
                                                   ClauseParam( design_entity: DesignEntity(DesignEntityType::READ, "r")),
                                                   ClauseParam( design_entity: DesignEntity(DesignEntityType::VARIABLE, "v1")));
  PatternClause pattern_clause = PatternClause(DesignEntity(DesignEntityType::ASSIGN, "a"),
                                               ClauseParam( design_entity: DesignEntity(DesignEntityType::VARIABLE, "v")),
                                               ClauseParam( design_entity: DesignEntity(DesignEntityType::WILDCARD, "_")));

  valid_query.AddClause(Clause(uses_clause));
  valid_query.AddClause(Clause(follows_clause));
  valid_query.AddClause(Clause(modifies_clause1));
  valid_query.AddClause(Clause(modifies_clause2));
  valid_query.AddClause(Clause(pattern_clause));

  QueryResult final_result = QueryEvaluator::EvaluateQuery(valid_query, &pkb, false);
  THEN( desc: "Select a such that Uses(a, v) and Follows*(a, s) and Modifies(s, v1) and Modifies(r, v1) pattern a(v, _) returns {5}") {
    REQUIRE(final_result.statement_indexes_or_constants.size() == 2);
    REQUIRE(Contains(final_result.statement_indexes_or_constants, 5));
    REQUIRE(Contains(final_result.statement_indexes_or_constants, 11));
  }
}
```

Selection tests

- Select BOOLEAN
- Select BOOLEAN where query with **clauses evaluates to true**
- Select BOOLEAN where query with **clauses evaluates to false**
- Select **DE** where DE was not evaluated in any clause and **retrieved from the PKB**
- Select **DE** where DE was evaluated in a clause and the **result space is different from that of the PKB**
- Select **attribute** where DE associated with attribute was **not evaluated in any clause**
- Select **attribute** where DE associated with attribute was **evaluated in some**

clause

- Select attribute where **attribute requires conversion** from its native type (e.g `print.varName` requires conversion from statements to variable names)
- Select tuple where all DEs in the tuple are **not evaluated in any clause** and retrieved from the PKB
- Select tuple where **some DEs** in the tuple are not evaluated in any clause and retrieved from the PKB
- Select tuple where one of the DEs **do not have any valid results** after evaluation of clauses return empty result
- Select tuple where some DEs are **completely identical** e.g `Select <a, a>`
- Select tuple of attributes where DEs associated with **each attribute were not evaluated in any clause**
- Select tuple of attributes where DEs associated with **each attribute were all evaluated in some clause**
- Select **tuple of DEs and attributes**
- Select tuple of **different attributes for the same DE** e.g `Select <pn, pn.varName, pn.stmt#>`
- Select tuple with **identical attributes** e.g `Select <a.stmt#, a.stmt#>`

Example of test cases

Purpose of Test Case: To test the selection of tuples with the same synonym but different attribute type

Required Input: A Query that selects a tuple of print, and the two associated attribute types of print, varName nad stmt#.

Expected Output: There should be no cross join between results of the pn, all values should correspond to the print statements in the source code in PKBStub.

```
WHEN( desc: "Tuple contains attributes of similar design entity") {
  DesignEntity pn = DesignEntity(DesignEntityType::PRINT, "pn");
  Query valid_query = Query(vector<SelectedEntity>{SelectedEntity(pn),
                                                   SelectedEntity(make_pair(pn,  t2: AttributeType::STMT_NO)),
                                                   SelectedEntity(make_pair(pn,  t2: AttributeType::VAR_NAME))});
  QueryResult final_result = QueryEvaluator::EvaluateQuery(valid_query, &pkb, false);

  THEN( desc: "Select <pn, pn.stmt#, pn.varName> returns <7, 7, sum> and <10, 10, x>") {
    REQUIRE(final_result.tuple_result.size() == 2);
    REQUIRE(Contains(final_result.tuple_result, vector<string>{"7", "7", "sum"}));
    REQUIRE(Contains(final_result.tuple_result, vector<string>{"10", "10", "x"}));
  }
}
```

Tests are conducted for each permutation of clause parameters, with DesignEntity being an overarching type (instead of individual design entities). For example, the unit tests for queries with a single follows clause might include the following test cases: Follows(INT, INT), Follows(INT, DE), Follows(INT, _), Follows(DE, INT), Follows(DE, DE), Follows(DE, _), Follows(_, INT), Follows(_, DE), Follows(_, _). For each category, a few valid and invalid test cases are brainstormed. The unit tests primarily aim to verify the basic functionality of clauses.

Beyond that, there are also some unit test cases designed to verify the ability of the Query Evaluator to evaluate two clauses (one such that and one pattern clause). These test cases are further divided by whether the clause evaluates independently (no overlapping synonyms), whether the clauses evaluate to true (both true, one true, both false), and whether the synonym selected is independent of the evaluated synonyms. For example, *Select s such that Follows(1, 2) pattern a(v, _)* includes two clauses that evaluate independently and a selected synonym that is distinct from the ones being evaluated. On the other hand, *Select a such that Modifies(a, v) pattern a(v, _"x"_)* involves the evaluation of two pairs of exactly identical synonyms, hence the clauses have to be evaluated dependently. The selected synonym also selects a previously evaluated synonym.

4.3 Integration Testing

The main interactions between each component in the SPA is as such:

Parser → Design Extractor: Passes a constructed AST from Parser to Design Extractor

Design Extractor → PKB: Insertion and retrieval of Design Abstractions in multiple passes

Query Evaluator → PKB: Retrieval of relevant information on certain Design Entities, Design Abstractions and other relationships

Query Parser → Query Evaluator: Passes a constructed Query object from Query Parser to Query Evaluator.

However, since the interactions between Parser and Design Extractor simply involve one output object becoming an input object for another, it is trivial to verify that the logic is correctly handled as long as the output objects and input objects in unit tests are verified for correctness. The same goes for Query Parser and Query Evaluator. Hence, the main logic interactions lie between Design Extractor and PKB, as well as QUery Evaluator and PKB, where multiple different API calls were made. Hence, integration testing was only done for Design Extractor-PKB and Query Evaluator-PKB.

4.3.1 Design Extractor-PKB

The Design Extractor-PKB tests generally test the correct extraction and population of design entities and abstractions given an AST, that has already been validated by the source parser.

The input to the integration test is a program string. It is parsed using the source parser component and an instance of the PKB is populated with relations extracted by the DE on the output AST. Correctness is checked by querying the PKB instance to ensure that it contains exactly the expected relations, no more and no less.

Many of the more complex relations are transitive through call statements. Hence, the relations naturally partition themselves into 2 stages of increasing difficulty to test. First

is single procedure programs with no call statements allowed. Second is multiple procedure programs with call statements allowed. This is the structure that our integration tests follow.

The tests are split into single procedure tests and multiple procedure tests. Within the single procedure tests, the following are tested:

Extraction of Design Entities
● Correct extraction of all statement numbers ● Correct extraction of all procedure names ● Correct extraction of all variables ● Correct extraction of all constants ● Correct extraction of all entity types ● Correct extraction of all read statements ● Correct extraction of all print statements ● Correct extraction of all assignment statements ● Correct extraction of all if statements ● Correct extraction of all while statements ● Correct extraction of all call statements

Extraction of Design Abstractions
● Correct extraction of all Follows/Follows* relations ● Correct extraction of all Parent/Parent* relations ● Correct extraction of all UsesS relations ● Correct extraction of all ModifiesS relations ● Correct extraction of all Next/Next* relations ● Correct extraction of all Affects/Affects* relations

With the addition of multiple procedures and call statements to the SIMPLE grammar, the DE had to support extraction of procedure information and the Call/Calls* relations among those procedures.

Extraction of Call Relations
● Correct extraction of all call statements ● Correct extraction of all procedure identifiers ● Correct construction of Call relations ● Correct construction of Call* relations

After iteration 1, Uses and Modifies queries on procedure identifiers was added. Testing of these relations was done directly. The addition of call statements meant that Uses and Modifies for statement numbers could be called on call statements, and had to return the same values as if the Uses and Modifies was called on the procedure itself. As before, variables used / modified by call statements had to propagate upward in the Parent tree.

Extraction of Modifies interacting with Calls and Parent
● Extraction of all variables a procedure that doesn't contain call statements modifies ● Extraction of all variables a procedure that contains call statements modifies ● Extraction of all variables a call statement modifies, where the procedure called doesn't contain call statements ● Extraction of all variables a call statement modifies, where the procedure does contain call statements ● Extraction of all variables an if statement modifies, where the container statement contains call statements ● Extraction of all variables a while statement modifies, where the container statement contains call statements

Extraction of Uses interacting with Calls and Parent
• Extraction of all variables a procedure that doesn't contain call statements uses • Extraction of all variables a procedure that contains call statements uses • Extraction of all variables a call statement uses, where the procedure called doesn't contain call statements • Extraction of all variables a call statement uses, where the procedure does contain call statements • Extraction of all variables an if statement uses, where the container statement contains call statements • Extraction of all variables a while statement uses, where the container statement contains call statements

Example of Integration Test Cases

```
procedure proc {
     if ((i == 1) && (foo == bar)) then {
       if (j == 2) then {
         while (k == 3) {
           print a4;
         }
       } else {
         read b5;
       }
     } else {
       while (l == 6) {
         if (m == 7) then {
           a = 8 + 219;
         } else {
           print e9;
         }
         read c10;
       }
```

```
            i = 11;
          }
        }
```

Purpose: To test that variables used by the conditional expressions of the if and while container statements are correctly populated into the PKB, especially with triple nested containers and multiple variables. This is to facilitate pattern matching for if and while.
Required Input: An empty PKB and a valid AST according to the source code above.
Expected Output: Correctly populated PKB

```
WHEN( desc: "Design extractor extracts all While statements") {
  PKB pkb = PKB();
  pkb.ClearAllTables();
  design_extractor::DesignExtractor::ExtractDesigns( &: pkb, root);
  std::unordered_set<int> while_stmts = pkb.GetAllWhileStmts();
  std::unordered_set<std::string> while_vars_3 = pkb.GetVariablesUsedByWhileStmt(3);
  std::unordered_set<std::string> while_vars_6 = pkb.GetVariablesUsedByWhileStmt(6);

  THEN( desc: "PKB gets populated with While information") {
    REQUIRE(while_stmts.size() == 2);
    REQUIRE(Contains(while_stmts, 3));
    REQUIRE(Contains(while_stmts, 6));
    REQUIRE(while_vars_3.size() == 1);
    REQUIRE(Contains(while_vars_3, "k"));
    REQUIRE(while_vars_6.size() == 1);
    REQUIRE(Contains(while_vars_6, "l"));
  }
}
```

```
WHEN( desc: "Design extractor extracts all If statements") {
  PKB pkb = PKB();
  pkb.ClearAllTables();
  design_extractor::DesignExtractor::ExtractDesigns( &: pkb, root);
  std::unordered_set<int> if_stmts = pkb.GetAllIfStmts();
  std::unordered_set<std::string> if_vars_1 = pkb.GetVariablesUsedByIfStmt(1);
  std::unordered_set<std::string> if_vars_2 = pkb.GetVariablesUsedByIfStmt(2);
  std::unordered_set<std::string> if_vars_7 = pkb.GetVariablesUsedByIfStmt(7);

  THEN( desc: "PKB gets populated with if information") {
    REQUIRE(if_stmts.size() == 3);
    REQUIRE(Contains(if_stmts, 1));
    REQUIRE(Contains(if_stmts, 2));
    REQUIRE(Contains(if_stmts, 7));
    REQUIRE(if_vars_1.size() == 3);
    REQUIRE(Contains(if_vars_1, "i"));
    REQUIRE(Contains(if_vars_1, "foo"));
    REQUIRE(Contains(if_vars_1, "bar"));
    REQUIRE(if_vars_2.size() == 1);
    REQUIRE(Contains(if_vars_2, "j"));
    REQUIRE(if_vars_7.size() == 1);
    REQUIRE(Contains(if_vars_7, "m"));
  }
}
```

4.3.2 Query Evaluator-PKB

The Query Evaluator-PKB test generally concerns the correct conversion of a query clause to retrieve the correct data from the PKB. In the QE-PKB tests, there is more focus on the correctness of certain types of clauses and design abstractions rather than query evaluation logic. Similar to previous sections, INT refers to statement indexes, NAME refers to strings that might be variable names or procedure names, and DE refers to some synonym that represents a design entity (e.g s for stmt, pn for print).

Integration tests on the Query Evaluator-PKB were done using an instance of PKB that is manually populated based on a source code and not using the Design Extractor.

Single clause tests
For **every design abstraction** for SuchThatClauses, one valid and invalid query for each test case: ● SuchThatClause(INT/NAME, INT/NAME) ● SuchThatClause(INT/NAME, DE) ● SuchThatClause(INT/NAME, _) ● SuchThatClause(DE, INT/NAME) ● SuchThatClause(DE, DE) ● SuchThatClause(DE, _) ● SuchThatClause(_, INT/NAME) ● SuchThatClause(_, DE) ● pattern a(_, _) ● pattern a(NAME, _) ● pattern a(NAME, complete match) ● pattern a(NAME, partial match) ● pattern a(DE, _) ● pattern a(DE, complete match) ● pattern a(DE, partial match)

- pattern a(_, partial match)
- pattern a(_, complete match)

- pattern w(_,_)
- pattern w(NAME, _)
- pattern w(DE, _)
- pattern ifs(_, _, _)
- pattern ifs(NAME, _, _)
- pattern ifs(DE, _, _)

- with clauses where the DE associated with the attributes not evaluated in any other clause
- with clauses where the DE associated with the attributes are evaluated in some other clause

Multiple clause tests

Refer to the Query Evaluator unit tests for multiple clauses. Functionally, the tests aim to verify the correctness of the QE-PKB integration to handle multiple clauses, although it should not differ too greatly from that of the QE.

Selection

- Select each design entity from PKB without any clauses
- Select DE where DE was not evaluated in any clause, hence needs to be retrieved from the PKB
- Select tuple where some DEs in the tuple was not evaluated in any clause, hence needs to be retrieved from the PKB
- Select DE in a query that returned true but DE does not contain any data in the

PKB

Example of Integration Test Cases

`procedure main {`
`  while (x > 0) {`
`  while (y > 0) {`
`  y = y - 1;`
`  while (z > 0) {`
`  z = z - 1;`
`  }`
`  if (y == z) then {`
`  print y;`
`  } else {`
`  print z;`
`  }`
`  }`
`  }`
`  print x;`
`}`
	Some example queries:
	
	`Select pn such that Parent(1, 2)`
	`Select pn such that Parent(2, 1)`
	`Select a such that Parent(4, a)`
	`Select w such that Parent*(w, 5)`
	`Select pn such that Parent(w, pn)`
	`Select pn such that Parent*(w, pn)`
	`Select pn such that Parent(_, pn)`
	`Select s such that Parent(s, _)`
	`Select pn such that Parent(pn, _)`
	`Select pn such that Parent(ifs, _)`

Example Test Case: Purpose: To test that indirectly nested statements are correctly inserted into Parent* table and not Parent table in the PKB, and that the information can be correctly retrieved by the Query Evaluator. Required Input: Query object with a Parent clause such that the second parameter is indirectly nested in the first parameter. Expected Output: Empty result.	

```
WHEN( desc: "Parent(DE, DE) evaluates to false, second DE is deeply nested in first DE, selected DE exists") {
  Query valid_query = Query(SelectedEntity( de: DesignEntity(DesignEntityType::STMT, "s")));
  SuchThatClause parent_clause = SuchThatClause(DesignAbstraction::PARENT,
                                                 ClauseParam( design_entity: DesignEntity(DesignEntityType::WHILE, "w")),
                                                 ClauseParam( design_entity: DesignEntity(DesignEntityType::PRINT, "pn")));
  valid_query.AddClause(Clause(parent_clause));
  QueryResult test_result = QueryEvaluator::EvaluateQuery(valid_query, &pkb_nested, false);
  THEN( desc: "Select s such that Parent(w, pn) returns empty result") {
    REQUIRE(test_result.statement_indexes_or_constants.empty());
  }
}
```

4.4 System Testing

For system testing, test cases were generally split into two types, one that focuses on the correctness of design extraction, hence tricky source code, and the other focuses on the correctness of evaluating queries. Some of the query test cases also test the efficiency of the query evaluation by evaluating on a large source code. Generic source code was generated by a SIMPLE source code generator.

Note: test suites 1, 3, and 5 within the main Tests21 directory require the NextBip/* and AffectsBip/* extensions to be turned on in order to pass the queries. Refer to the GitHub repository's README to find out how to toggle these extensions on.

4.4.1 Testing of Source Code Parsing and Design Extraction

Tests for source code parsing and design extraction focused on complex source codes, and verified that the SourceProcessor and DesignExtractor combined were able to identify valid source code, parse it, and correctly extract all design abstraction.

Since query processing was not the focus of this test suite, the queries used were simple, and usually consisted of one or two clauses regarding the design abstraction under test.

Test Suite Components

Purpose: To test the behaviour of SPA when met with an invalid SIMPLE source. In this case, SPA should exit with error code 1, print the error messages from the Parser, and should not proceed with any query evaluation.
Explanation: The Parser should be able to properly identify an invalid source and show any appropriate error messages to the user. In the example source code below, since conditional expressions cannot be surrounded by arbitrary parentheses, this is an invalid source code due to statement 3.

Example source code:
`procedure CondExprCannotHaveArbitraryParentheses {`
`  while ((1) == (1)) {`
`  IsAboveCondExprValid = 1;`
`  }`
`  while ((1 == 1)) {`
`  IsAboveCondExprValid = 0;`
`  }`
`}`

Example queries: irrelevant

For Design Extractor tests, the focus is on new design abstractions including Calls/*, Next/*, Affects/*, NextBip/* and AffectsBip/* and thus the source code is designed to focus on a high level of nesting, multiple Call and multiple Assignment statements. Similarly, the queries are more centered on testing the correct construction of the CFG, the CFGBip as well as whether complex queries that need to satisfy multiple clauses with a combination of both new and old relationships can be handled.

Purpose:

To test the Design Extractor's construction of the CFG and extraction of other relationships.

Explanation:

The Design Extractor should be able to construct the correct CFG despite a high level of nesting as well as correctly extract all other relationships within a single procedure. Below are the relationships and clauses mainly focused on:

- Next/Next* from the CFG constructed
- Affect/Affect* from the CFG constructed
- Parent/Parent* due to high level of nesting
- Calls/Calls*
- Modifies and Uses due to the focus on assignment statements
- Pattern clauses due to the focus on assignment and container statements

Example source code: (Shortened for illustration purpose only)

```
procedure P3 {
 ...
 if (v6 > v1 + v5 * (v5)) then {
   if (v4 > 96) then {
     read v1;
     while (354 != v8 + v10) {
       read v8;
     }
   } else {
     while (v4 >= 631) {
       v1 = v1 - v6 * v7 + v9 * v10 - v4 % (v6) % v10 * v7;
       if (v9 != v8) then {
         while (v4 < v1) {
           print v5;
```

```
        }
      } else {
        call P1;
        while (330 > v10) {
          print v3;
        }
      }
    }
  }
  ...
} else {
  if (862 == v5 - v5 / v8) then {
    while (901 != v6) {
      ...
      while (959 == v2) {
        print v10;
        while (v4 != v4) {
          v8 = v10 + v10 / (677) - v6 + v10 / 34;
        }
      }
    }
    ...
}
```

Example queries:

11 - Find whether there is an execution path from statement #48 to statement #75 that passes through #59

```
Select BOOLEAN such that Next*(48, 59) and Next*(59, 75)
TRUE
5000
```

17 - Find all assign statements in between an execution path from 18 to 69

```
assign a;
Select a such that Next*(18, a) and Next*(a, 69)
32, 50, 54, 61, 62, 66
5000
```

18 - Find all assign statements in between an execution path from 18 to 69 that stmt# 50 Affects

```
assign a;
Select a such that Next*(18, a) and Next*(a, 69) and Affects(50, a)
50, 54, 61, 62
5000
```

19 - Find all pairs of variable and assignment combined that satisfy all conditions

```
assign a; if ifs; while w; variable v; constant c; procedure p;
Select <v, a> such that Next*(18, a) and Next*(a, 75) and Affects(40, a) pattern ifs(v, _, _) and w(v, _) and a("v1", _) with a.stmt# = c.value
v1 40, v2 40, v3 40, v4 40, v6 40, v7 40, v8 40, v9 40, v10 40, v5 40
5000
```

35 - Find all while, statement, assignment, variable tuples that satisfy all conditions

```
while w; print pn; stmt s; variable v, v1; assign a, a1;
Select <w, s, a1, v1> such that Parent*(w, s) and Uses(s, v) and Affects(s, a) and Next*(s, pn) and Uses(pn, "v2") and Next*(27, a1) and Modifies(a1, v1) and Uses(a1, "v5")
77 80 50 v10, 77 80 75 v2, 123 128 50 v10, 123 128 75 v2, 132 133 50
```

v10, 132 133 75 v2
5000

36 - Find all assign statements deeply nested in 4 levels of ifs and directly nested in the last layer, query tuple

if if1, if2, if3, if4; assign a;
Select <if1, if2, if3, if4, a> such that Parent*(if1, if2) and Parent*(if2, if3) and Parent*(if3, if4) and Parent(if4, a)
131 134 135 136 138, 34 48 55 56 61, 34 48 55 56 62, 34 48 55 63 66
5000

37 - Find all assign statements deeply nested in 3 levels of ifs and directly or indirectly nested in the last layer that Affects some other assign statement, query tuple

if if1, if2, if3; assign a, a1;
Select <if1, if2, if3, a, a1> such that Parent*(if1, if2) and Parent*(if2, if3) and Parent*(if3, a) and Affects(a, a1)
34 48 55 62 75, 34 48 56 62 75, 34 55 56 62 75, 48 55 56 62 75
5000

38 - Find all assign statements deeply nested in 3 levels of ifs and directly or indirectly nested in 1 level of while that Affects some other assign statement, query tuple

while w1, w2, w3; assign a, a1;
Select <w1, w2, w3, a, a1> such that Parent*(w1, w2) and Parent*(w2, w3) and Parent*(w3, a) and Affects(a, a1)
49 51 53 54 50
5000

39 - Find all pair of assign statements nested in 3 levels if, if, while and a

Affects* a1

```
if if1, if2; while w; assign a, a1;
Select <a, a1> such that Parent*(if1, if2) and Parent*(if2, w) and
Parent*(w, a) and Affects*(a, a1)
40 40, 40 47, 40 75, 50 50, 50 54, 50 61, 50 62, 50 75, 54 50, 54 54,
54 61, 54 62, 54 75
5000
```

Purpose:

To test the Design Extractor's construction of the CFGBip and extraction of other relationships.

Explanation:

The Design Extractor should be able to construct the correct CFGBip despite a high level of nesting and multiple Call statements as well as correctly extract all other relationships across multiple procedures. Below are the relationships and clauses mainly focused on:

- NextBip/NextBip* from the CFGBip constructed
- AffectBip/AffectBip* from the CFGBip constructed
- Parent/Parent* due to high level of nesting
- Calls/Calls*
- Modifies and Uses due to the focus on assignment statements
- Pattern clauses due to the focus on assignment and container statements

Example source code:

Source code is not important in understanding the purpose of these test cases since the focus is on the queries themselves.

For the actual source code, refer to

Tests21/iteration3/8_relationship_emphasis_source.txt

23 - Find all while statements on NextBip execution path from stmt #76 such that it contains some read statements

```
while w; read r; assign a;
Select w such that NextBip*(76, w) and Parent*(w, r)
77, 85, 12, 92
5000
```

24 - Find all if statements on NextBip execution path from stmt #76 such that it uses variable 'v8' in conditional block and uses 'v2'

```
if ifs;
Select ifs such that NextBip*(76, ifs) pattern ifs("v8", _, _) such
that Uses(ifs, "v2")
4, 11, 79, 86, 90
5000
```

25 - Find whether there are some assign statements which are Affected by statement 76 and satisfy all remaining clauses

```
assign a;
Select BOOLEAN such that Affects*(76, a) and Affects*(80, 81) and
Affects*(76, 96) and Affects(76, 96) and Affects(80, 81)
TRUE
5000
```

26 - Find previous statements of stmt #106 within a procedure and previous statements of stmt #110 across procedures

```
stmt s; prog_line n;
Select <s, n> such that Next(s, 106) and NextBip(n, 110)
105 74, 112 74, 118 74, 117 74
```

5000

208 - All assign statements that affect another statement and uses and modifies the same variable and is contained in a while that uses that variable in its conditional expression

```
variable v; assign a; while w;
Select <a, v> such that AffectsBip(a, _) and Uses(a, v) and
Modifies(a, v) and Parent*(w, a) pattern w(v, _)
108 v3,128 v10
5000
```

209 - All assign statements that affect another statement and uses and modifies the same variable and is contained in a if that uses that variable in its cond exp

```
variable v; assign a; if ifs ;
Select <a, v> such that AffectsBip(a, _) and Uses(a, v) and
Modifies(a, v) and Parent*(ifs, a) pattern ifs(v, _, _)
133 v10,32 v3,40 v1,62 v10,7 v6,81 v8
5000
```

213 - Find all Affectsbip* statements affected by statement 101 across procedures and is contained in a while loop directly or indirectly

```
assign a; while w;
Select a such that AffectsBip*(101, a) and Parent*(w, a)
80, 81, 89, 96, 97
5000
```

214 - Find all Affectsbip* statements affected by statement 101 across procedures and is contained in a if block indirectly

```
assign a; if ifs; stmt s;
Select a such that AffectsBip*(101, a) and Parent*(ifs, s) and
```

```
Parent(s, a)
96, 97
5000
```

Purpose:

To test the Parser on valid source code that has some edge cases.

Explanation:

The Parser should be able to properly parse a program that have the following edge cases:

- Using reserved keywords as names
- Arbitrary parentheses within a mathematical expression

Example source code:

```
procedure main {
  while (((WHILE + (PRINT007)) != read) || ((if) <= else)) {
    if (!((420 > then) && (print < while))) then {
      read read;
    } else {
      while (((((while)))) >= (((69) + ((then))) % call)) {
        print = x1 + 69;
      }
    }
  }
}
```

Example queries:

6 - Valid Parent(INT, INT) returns all variables

Test that all variables are correctly parsed as variables

```
variable v;
Select v such that Parent(1, 2)
WHILE, PRINT007, read, if, else, then, print, while, call, x1
5000
```

4.4.2 Testing of Query Processing

Tests for query processing generally involved complex queries, meant to test the ability of the Query Parser and Query Evaluator to parse and correctly evaluate complex queries. Additionally, tests for optimization were also included, to test the ability of the Query Optimizer to re-order the clauses in the query based on chosen heuristics.

Since the extraction of the design abstractions in the source code is not the focus of this test suite, the source code is not designed to be very complex. The source code was randomly generated using a Python script, and queries were chosen carefully based on the existing source code.

Test Suite Components

Purpose: Correct invalid query handling of Query Parser
Explanation: There are two distinct return values for invalid queries - if the query is syntactically valid, has a semantic error, and selects BOOLEAN, then FALSE should be returned; otherwise, nothing should be returned instead. These test cases were used to verify that the Query Parser correctly recognized syntactic and semantic errors, and returned the correct return value.
Example source code:

```
procedure main {
	thisSourceCodeIsIntentionallyDesignedToBeCorrect = 0;
	theWholePointIsThatNoneOfThisMattersSinceAllQueriesShouldFail =
1;
	youCanConsiderThisAnEasterEggIfYouFoundIt = 2;
	howeverWeHaveToAddSomeStatementsHere =
soThatTheyCanBeReturnedIfSuccessSuddenlyOccurs;
	thereIsOneDesignEntityTypeOfEverything = inThisSourceCode;
	CanAnybodyPleaseOpenOur =
CalculationsAndProfileOutOptimizations;
	read r;
	print pn;
	call test;
	if (1 == 1) then {
		a = 1;
	} else {
		a = 1;
	}
	while (1 == 1) {
		a = 1;
	}
}

procedure test {
	a = 1;
}
```

Example queries:

95 - Test for selection of tuples with extra comma

The additional comma in the selected tuple is considered a syntactic error. This is to test that the Query Parser identifies that a comma must be followed by a term.

```
stmt s, s1, s2;
Select <s, s1, s2,>
none
5000
```

102 - Test for no whitespace between 'BOOLEAN' and 'such that'

Missing whitespace should be considered a syntactic error, since 'BOOLEAN' and 'such' are both alphabetical. 'Select BOOLEAN' is used to ensure that this error is considered syntactic.

```
Select BOOLEANsuch that Follows(_, _)
none
5000
```

476 - Test for assign pattern expression containing incomplete parenthesis grouping

An invalid pattern expression in assign pattern statements should be considered a syntactic error. 'Select BOOLEAN' is used to ensure that this error is considered syntactic.

```
assign a;
Select BOOLEAN pattern a(_, "x * (( y + z)")
none
5000
```

1804 - Test for with clause with wrong LHS and RHS combination; synonym selected

The LHS of the with clause returns a string but the RHS returns an integer, which is not allowed. 'Select a' is used to ensure that this error is considered semantic, but nothing is returned because BOOLEAN was not selected.

```
assign a; variable v;
Select a with v.varName = a.stmt#
none
5000
```

1854 - Test for with clause with wrong LHS and RHS combination; BOOLEAN selected

The LHS of the with clause returns a string but the RHS returns an integer, which is not allowed. 'Select BOOLEAN' is used to ensure that this error is considered semantic, and FALSE is expected.

```
assign a; variable v;
Select BOOLEAN with v.varName = a.stmt#
FALSE
5000
```

Purpose:

Selection correctness of the query processor

Explanation:

This set of test cases try to test the correctness of selection. Most of these queries either select from the PKB without any clauses, or independent selection of PKB even after evaluating clauses.

Source Code:

Source code is not important in understanding the purpose of these test cases since the focus is on the queries themselves.

For the actual source code, refer to

`Tests21/iteration3/4_complexqueries_source.txt`

Example queries:

17 - Select print.varName

Test selection of attribute that is different from the design entity's original form.

```
print pn;
Select pn.varName
a, b, c, d, f, g, h, i, j, cap00
5000
```

24 - Select tuple single synonym

```
print pn;
Select <pn>
2, 8, 13, 19, 22, 31, 36, 37, 38, 43, 44, 45, 54, 58, 60, 67, 76, 81,
85, 91, 93, 102, 103, 108, 121, 139, 147, 159, 161, 164, 166, 168,
170, 173, 175
5000
```

26 - Select tuple same synonym

Test repeated synonyms in tuple selection.

```
assign a;
Select <a, a>
3 3, 7 7, 16 16, 23 23, 27 27, 28 28, 29 29, 32 32, 33 33, 39 39, 46
46, 47 47, 49 49, 50 50, 59 59, 73 73, 74 74, 79 79, 82 82, 89 89, 90
90, 96 96, 106 106, 113 113, 117 117, 118 118, 119 119, 120 120, 125
```

```
125, 126 126, 131 131, 136 136, 142 142, 143 143, 145 145, 146 146,
149 149, 174 174
5000
```

121 - With clauses, independent evaluation, independent selection, return true
Test selection of synonyms that have not been evaluated.

```
print pn; call cl; stmt s; assign a;
Select a with 1 = 1 and "a" = "a" and "cap00" = pn.varName and
cl.procName = "cap00" and s.stmt# = 14
3, 7, 16, 23, 27, 28, 29, 32, 33, 39, 46, 47, 49, 50, 59, 73, 74, 79,
82, 89, 90, 96, 106, 113, 117, 118, 119, 120, 125, 126, 131, 136,
142, 143, 145, 146, 149, 174
5000
```

Purpose:

Correctness of the evaluation of complex queries

Explanation:

These test cases try to test the correctness of the merging of tables on queries that have many clauses, with different groups. Test cases are done incrementally, starting from multiple SuchThat Clauses, multiple PatternClauses, multiple WithClauses, followed by combinations of multiple SuchThatClauses and PatternClauses, so on and so forth. For each combination, queries with independent clauses and dependent clauses that return true or false were done.

Example source code:

Source code is not important in understanding the purpose of these test cases since the focus is on the queries themselves.

For the actual source code, refer to

Tests21/iteration3/4_complexqueries_source.txt

Example queries:

105 - Such that clauses with identical parameters, result true

```
assign a; stmt s;
Select a such that Next*(a, a) and Affects*(a, a) and NextBip*(s, s) and Affects(s, s)
32, 33, 96, 131
5000
```

107 - Such that clause with overlapping parameters, dependent tuple selection, result true

```
while w; print pn; stmt s; variable v, v1; assign a, a1;
Select <w, pn, a1, v1> such that Parent(w, s) and Uses(s, v) and Next*(s, a) and Follows(s, pn) and Follows*(27, a1) and Modifies(a1, v1)
4 38 28 b, 5 13 28 b, 15 19 28 b, 34 37 28 b, 51 54 28 b, 55 60 28 b, 78 81 28 b, 171 175 28 b, 4 38 29 c, 5 13 29 c, 15 19 29 c, 34 37 29 c, 51 54 29 c, 55 60 29 c, 78 81 29 c, 171 175 29 c
5000
```

131 - Such that + pattern clauses, independent evaluation, independent selection, return true

All clauses do not have any overlapping synonyms, and selection is done straight from the PKB.

```
constant c; prog_line n; assign a, a1; variable v, v1; while w; if ifs;
Select c.value such that Next(176, n) such that Affects(a, 174) pattern a1(v, _"e%e%c"_) pattern w(v1, _) pattern ifs("a", _, _)
```

```
pattern ifs("g", _, _)
9, 15, 21, 25, 66, 77, 84, 86, 92, 94, 100, 102, 119, 133, 142, 183,
184, 205, 213, 214, 218, 220, 221, 263, 268, 270, 275, 279, 285, 308,
312, 331, 332, 349, 368, 388, 392, 395, 418, 424, 429, 431, 432, 433,
445, 450, 451, 457, 469, 484, 512, 516, 520, 525, 546, 547, 550, 551,
552, 562, 573, 583, 592, 630, 631, 633, 636, 650, 657, 681, 684, 698,
707, 744, 753, 764, 769, 772, 773, 774, 809, 829, 834, 842, 843, 845,
850, 853, 863, 894, 897, 900, 906, 909, 913, 920, 922, 927, 931, 945,
969, 984, 985, 993, 997, 998
5000
```

154 - Pattern + with clauses, dependent evaluation, return false

```
while w; variable v, v1; if ifs; call cl;
Select BOOLEAN pattern w(v, _) pattern ifs(v1, _, _) with v.varName =
v1.varName with v1.varName = cl.procName
FALSE
5000
```

Purpose:

Effectiveness of Query Optimizations

Explanation:

There are some queries which were determined to be difficult to evaluate naively based on the Query Evaluator's implementation without any optimizations. These test cases were used to verify that the optimizations are effective in reducing time taken for these clauses, and do not have any regressive effect on other queries.

Example source code:

Source code is 500 lines long. The length of the source code itself is the main test feature, therefore no one specific part can be used as an example.
Refer to `Tests21/iteration3/5_optimization_source.txt`

Example queries:

3 - Test removal of repeated clauses
Repeated clauses will be removed, hence significantly decreasing the evaluation time

```
assign a1;
Select BOOLEAN pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _) pattern a1(_, _) pattern a1(_, _) pattern a1(_, _)
pattern a1(_, _)
TRUE
5000
```

4 - Test sorting of clauses by number of synonyms
The with clause should be sorted and evaluated first, avoiding evaluation of a large inner join for Next*

```
call cl; stmt s, s1, s2, s3;
```

```
Select cl such that Next*(s1, s2) and Next*(s2, s3) with 1=2
none
5000
```

6 - Test that only related clauses are merged

The clauses below all do not have any overlapping synonyms, hence no cross products should be performed, saving significant evaluation time.

```
call cl; stmt s, s1, s2, s3, s4, s5, s6, s7, s8, s9;
Select cl such that Follows(s, s1) and Follows(s2, s3) and Follows
(s4, s5) and Follows (s6, s7) and Follows (s8, s9)
111,114,128,13,130,131,137,14,152,155,158,181,184,185,207,210,224,230
,234,244,246,262,266,273,283,296,304,318,321,323,341,346,354,356,373,
392,42,45,54,55,71,73,77,92,93
5000
```

Purpose:

Stress test the cross join algorithm and output of the Query Evaluator

Explanation:

This test case trivially selects two design entities with very large outputs in a tuple. The two design entities will be cross joined to form a table of > 100k lines to stress test the cross join algorithm.

Example source code:

The source code contains 636 variables and 237 constants and can be found under `Tests21/stress/1_source.txt`

Query:

2 - Select large tuple (636 * 237 = 150732)

The query selects a tuple that returns 150k lines.

variable v; constant c;

Select <v, c>

The results are not shown since it is too long and is not necessary to understand the test case.

5000

4.5 Test Utilities

4.5.1 Test Runner Script

For iteration 2, a helper script using Python was written to run all unit, integration, and autotester tests. This helper script will automatically check that every autotester test case passes. Since there are a substantial number of test cases, this eliminates the need to run each of them individually and checking each `out.xml` file manually, thereby making running of tests much more efficient.

4.5.2 SIMPLE Source Code Generator

For iteration 3, a Python CLI-based script was made which can generate a random and valid SIMPLE source code. Parameters could be passed via the CLI to control the number of procedures, statements, variables and constants, as well as control how many nesting levels container statements should have or how long an expression can possibly be, etc. The full instruction of how to use the script can be seen on the GitHub repository's README.

5 Extensions to SPA

5.1 Implementation Schedule

The development plan for the extension is as follows:

	Week 10	**Week 11**	**Week 12**
Source Parser (Aaron)	Enhance efficiency of lexer to deal with longer source codes	CLI-based Python script to generate random SIMPLE source code for stress and correctness testing	Creation of the CFG explosion for use to extract design entities for AffectsBip /AffectsBip*
Design Extractor (Jia Da and Zi Ying)	Population of Next/Next*Affects/ Affects* relationship	Population of NextBip/NextBip* relationship	Population of AffectsBip/AffectsBip* relationship
PKB (Hung)	Storage and APIs for Affects/ Affects*	Storage and APIs for NextBip/NextBip*	Storage and APIs for AffectsBip/AffectsBip*
Query Evaluator (Jolyn)	Next/Next*/Affects/ Affects* queries	NextBip/NextBip* queries, and improving efficiency of Query Evaluator	AffectsBip/AffectsBip* queries
Query Parser (Samuel)	Next/Next*/Affects/] Affects* queries	NextBip/NextBip* queries, and	AffectsBip/AffectsBip* queries

		improving efficiency of Query Parser	
Testing (Aaron)			[H, JD, ZY] Creation of complex DE source programme (deep nesting)

5.2 Implemented Changes

This section describes the changes implemented to support the extraction of NextBip/NextBip*/AffectsBip/AffectsBip*. These four design abstractions have their PQL concrete grammar defined the same as Next/Next*/Affects/Affects*.

First, two approaches for constructing the CFGBip are discussed. Then follows a discussion about algorithms used to extract the relations from the respective CFGBip graphs. It ends off with the test plan for extensions, and finally relevant design decisions. The final decision was to implement both methods of constructing the CFGBip, with NextBip/NextBip* using a version similar to the “labelled edges CFGBip” in the Wiki, and AffectsBip/AffectsBip* using a version similar to the “graph explosion method” in the Wiki.

5.2.1 CFGBip

In this discussion, the terminology of BranchIn and BranchBack edges will follow that of the Wiki. The extraction of the NextBip and NextBip* design abstractions rely on the use of a CFGBip. It is similar to the Generalized CFGBip with labeled edges shown in the wiki, with 2 differences. First, dummy nodes are not used. Secondly, the BranchBack edges are not labelled.

A CFGBipHandler was created for the construction of the CFGBip. It requires that the CFG is already constructed. As it traverses the CFG, if it encounters a call statement, it

will create a BranchIn edge from that call statement to the first statement node of the called procedure. Then it will remove the Next edge from the call statement to its Next statement, if it exists. Lastly, it will connect all the exit statements of the called procedure (which may not lie within the called procedure) to the Next statement of the call statement, if it exists. For every procedure, an exit statement is any statement that could possibly be the last executed statement before execution leaves that procedure.

Note that since dummy nodes are not used to unify a single exit node for each procedure, a procedure might have more than one exit node. Further, if a call statement is also an exit statement of a procedure, the source of the BranchBack edge could be in the called procedure, or in a procedure the called procedure subsequently calls.

In this construction, BranchIn and BranchBack edges are indistinguishable from regular Next edges. Further, since dummy nodes are not added, the correspondence between the nodes of the CFGBip and statement numbers of the source program is maintained. Both of those together allow the CFGBip to be stored as a simple adjacency list, which provides some convenience when extracting NextBip relation (discussed later).

5.2.2 CFG Explosion

The extraction of the AffectsBip and AffectsBip* design abstractions rely on the use of an exploded CFG. Similar to what is shown in the Wiki, a new instance of a graph for procedure P at each place where P is called is embedded to the caller node, and invalid edges removed. However, in our implementation, there are no dummy nodes (ie. nodes connect directly without first connecting to any dummy nodes).

For every SIMPLE program, the CFG explosion is a graph, with one or more disjoint components. Thus, the root of such a disjoint component can be thought of as a procedure A which is not called by any other procedure B (since if it is called by procedure B, then B’s nodes must be preceded by procedure A’s nodes). The exploded graph is represented using a vector of GENodes (which stands for graph explosion node), where each GENode in it represents the root node of one single component produced by the SIMPLE program.

The creation of such a CFG explosion relies on both the previously created AST and CFG, and is created from a top down approach using recursion. Firstly, all possible root procedures are identified. Then, assume that there exists a method called ExplodeGraph which can return the CFG explosion of a given procedure. In this method, while going through the normal CFG of a procedure, whenever a call statement is met, the ExplodeGraph method can be called recursively to explode that procedure's CFG and connect to it appropriately. If any statement other than a call statement is encountered, it will be connected exactly like how the original CFG describes.

If there exists a statement E after the call statement, the last evaluated statement in the called procedure must point back to statement E. Since the exploded CFG is built top down in a recursive manner, the recursion stack of ExplodeGraph can be used to "remember" which exit statement each call to ExplodeGraph should eventually point to.

An edge case for the previous step exists where the last statement of the called procedure is an If statement, since both the last statements of the Else and Then statement lists must be considered. To handle this, another recursive function is used to recursively search for and connect the last statements of that If statement (since the last statement of an If statement can be another If statement).

5.2.2.1 Example CFG Explosion

Assume we have the following SIMPLE program, with procedures A and B:

```
procedure A {
  x = 1;              // 1
  if (x == 2) then { // 2
    call B;           // 3
  } else {
    y = 4;            // 4
  }
  call B;             // 5
}

procedure B {
  if (x == 6) then { // 6
    read b;           // 7
```

```
  } else {
    read c;          // 8
  }
}
```

The original CFGs of both procedures would thus look like the following:

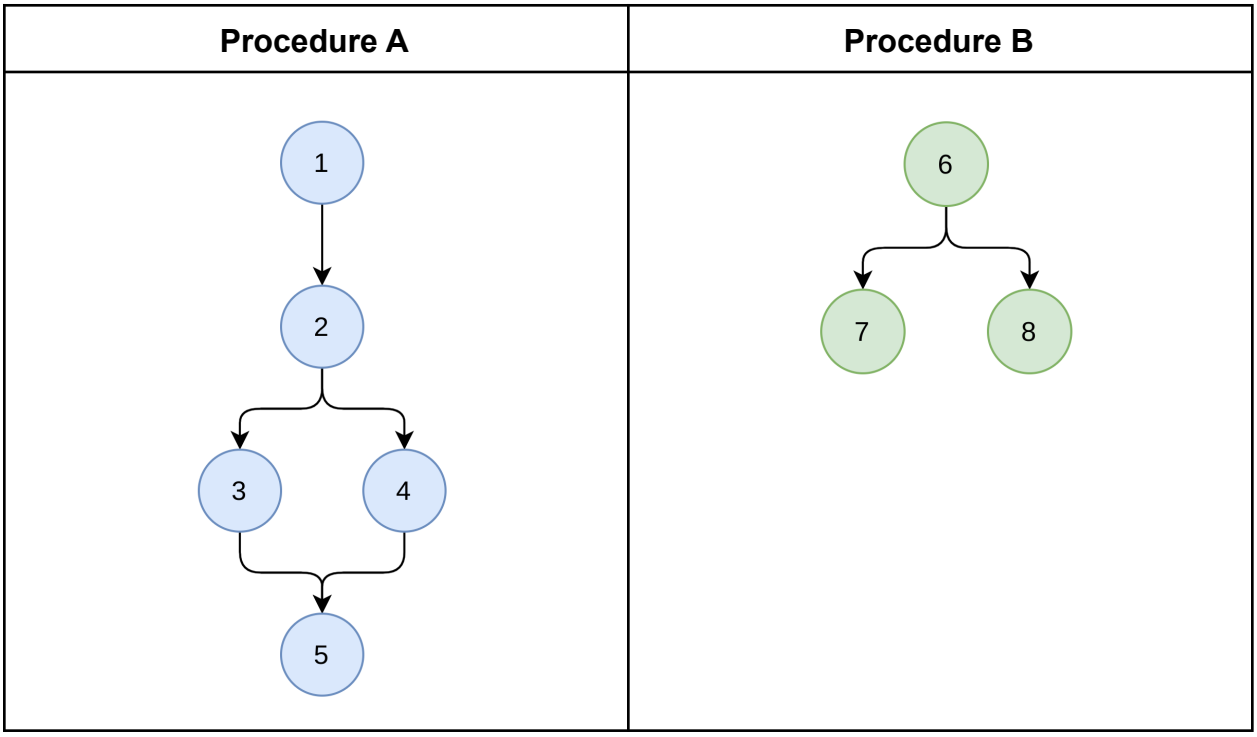

Figure 5.2.2.1.1: Original CFG For Procedures A and B

Following the algorithm as laid out in the previous section, every call statement must be connected to a *new* instance of the called procedure, and that called procedure's last executed statement(s) must connect back to the caller procedure's next statement after the call statement (if any). The following is then the exploded CFG:

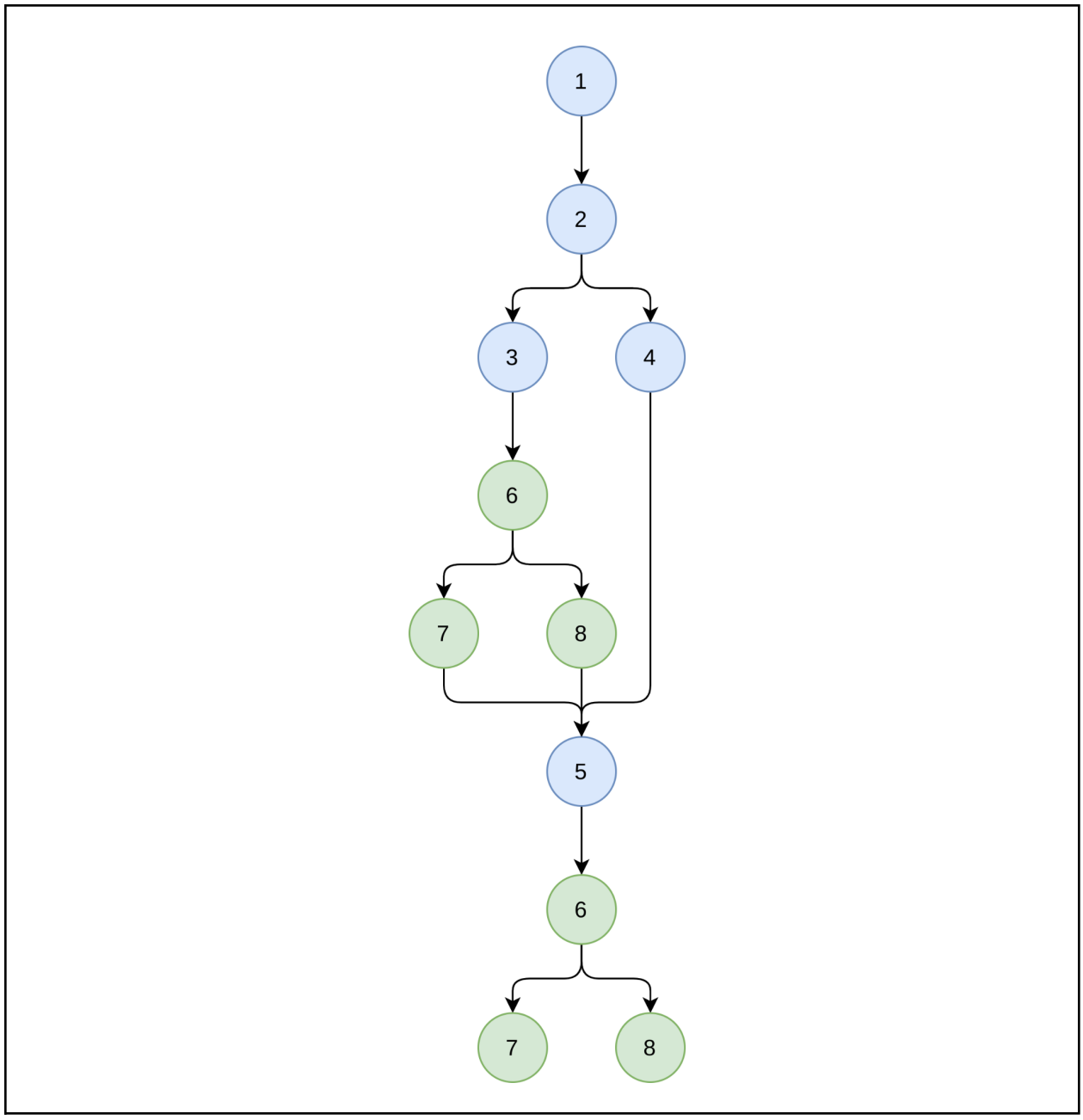

Figure 5.2.2.1.2: Exploded CFG For Procedure A

In particular, note how the first call statement at node 3 has connections from node 7 and 8 back to node 5, but the last call statement at node 5 does not. In the C++ implementation, nodes 6, 7, and 8 are instantiated twice, once for each call statement, and are differentiated by their memory pointers. Node 1 is thus the root of this exploded CFG, and will be saved as an entry in the vector of GENodes which is passed to the AffectsBipHandler to extract the AB and AB* design entities.

5.2.3 Design Extractor

5.2.3.1 NextBip

With the modified construction of CFGBip, it can be stored as a simple adjacency list. This is because there is no addition of dummy nodes or duplicate nodes as in the graph explosion method. Hence, each node corresponds to exactly 1 statement in the source program. Also, since there are no dummy nodes, each edge corresponds to exactly a single NextBip relation. In other words, NextBip(a,b) is true if and only if there is an edge from node a to node b in CFGBip. Hence, the algorithm to extract the NextBip relation by iterating over all edges of the CFGBip and populating the PKB.

5.2.3.2 NextBip*

NextBip*(a,b) is true if and only if when starting traversal from node a in the CFGBip, node b is reachable. Hence, for every node s in the CFGBip, we do a DFS traversal using that node as the source, keeping track of all reachable nodes. At the end of each traversal, for all reachable nodes r, we populate the PKB with NextBip*(s, r).

The traversal is not a simple DFS (which maintains which nodes it has visited so far, and avoids visiting them more than once) because for a given node, traversal cannot always continue through all outgoing edges to unvisited nodes. This is because the traversal is “execution-aware”. When at an exit statement, if we are in the current procedure because we took a BranchIn edge, traversal can only continue through the BranchBack edge returning to the caller procedure. If there are no such BranchIn edges, then traversal continues through all BranchBack edges, continuing execution from the next statements after all possible call statements to the current procedure.

Further, we can visit a node more than once. Take the following source as an example.

```
procedure A {
     call B;    //1
     call B;    //2
     call B;    //3
```

```
}
procedure B {
      print var; //4
      print var; //5
      print var; //6
}
```

NextBip*(1,3) is true, but requires traversing through statements 4,5,6 twice, once for each call statement in line 1 and 2. On the other hand, not maintaining visited nodes cannot be allowed because the presence of while loops in the source code will cause cycles in the CFGBip, and hence infinite loops in the traversal. The final solution uses an “allowance” system. For every BranchIn edge traversed, every node in the called procedure is allowed to be visited one more time. This allows a procedure to be traversed as many times as it is called, yet prevents unnecessary intra-procedure traversing due to cycles in the CFGBip.

5.2.3.3 AffectsBip

For AffectsBip, the exploded CFG is used. As mentioned above, the exploded CFG is represented using a vector of GENodes, with the GENodes representing the root node to a connected exploded CFG graph. For each connected exploded graph, a breadth-first traversal is done, finding all GENodes that represent an assign statement. Each of these assign statement GENodes are then sent to a depth-first traversal method. This essentially is a brute force algorithm where all possible paths within the exploded CFG will eventually be explored.

While traversing the exploded graph, paths that have a read or assign statement that modifies the original modified variable will not be traversed any further. However, this does not stop the traversal of the alternative paths. This is in line with the definition of AffectsBip, where there has to exist an execution path where the variable is not modified by another assign or read statement.

However, this results in a nested graph traversal, which causes the AffectsBip algorithm to have a time complexity of $O(n^2)$, with n being the number of nodes in the exploded CFG. However, in the worst case, the number of nodes in the exploded graph is exponential with respect to the total number of statements, since every call statement will result in an addition of the number of statements in the called procedure. Therefore, in this worst case, the AffectsBip algorithm will be exponential with regards to the number of statements in the program.

Take the following source as an example, the assign statement 5 is able to affect statement 7 as there is an execution path where the variable “a” is not modified across the procedural calls. The same logical argument can be made for assign statement 6, which AffectsBip line 5. However, for line 7, once the traversal reaches line 2 after the first call of procedure B, it stops processing that path as the variable c is modified in line 2. Therefore line 6 will not be affected by line 7.

```
procedure A {
      call B;    //1
      read c;    //2
      call B;    //3
}
procedure B {
      d = a;     //4
      a = b;     //5
      b = c;     //6
      c = d;     //7
}
```

5.2.3.4 AffectsBip*

Extraction of AffectsBip* is reliant on AffectsBip to be extracted first, as it has to wait for the AffectsGraph to be populated, which is done while extracting AffectsBip. The

AffectsGraph is represented by the mapping of an assign statement GENode to a set of GENodes that are directly affected by that particular assign GENode. For every assign GENode, the set of affected GENodes are retrieved. Then a breadth-first traversal is done from the assign GENode, fetching all possible AffectsBip* statements, inserting all possible AffectsBip* relationships.

The time complexity of this is also $O(n^2)$, where n represents the number of assign GENodes in the exploded graph. Similar to AffectsBip, in the worst case, there may be many calls to the same procedure, which makes the exploded graph grow exponentially as it continuously replicates the called procedure's CFG. Therefore the AffectsBip* algorithm is potentially exponential with regards to the number of GENodes in the exploded graph. However, it does guarantee that all paths will be traversed, and produces all AffectsBip* relations deterministically.

5.3 Test Plan

Tests are broken up into 3 separate phases:

1. Testing that the CFGBip and exploded CFG are correctly constructed
2. Testing that the NextBip and NextBip* design entities are correctly extracted
3. Testing that the AffectsBip and AffectsBip* design entities are correctly extracted

For 1, the CFGBip and exploded CFG are tested in integration tests, with the only dependency on the Source Processor Parser. The output graphs are then checked against a hard coded graph.

For 2 and 3, these are also tested in integration tests together with the PKB, to ensure that the Query Processor will be able to eventually query for such design entities.

Construction of CFGBip and Exploded CFG
NOTE: For each of the following tests, a correct CFG is provided for a given

SIMPLE program. The constructed CFGBip / Exploded CFG is compared against the expected CFGBip / Exploded CFG for structural equivalence.

Types of test cases:

- Single procedure programs
- Multiple procedure, no call statement programs
- Procedures with call statements that have Next statements within the same procedure
- Programs with procedures that have call statements as exit statements of that procedure
- Programs with a linear call chain of procedures that have call statements as exit statements
- Programs with a call tree of procedures that have call statements as exit statements
- Programs with call statements nested within top-level container statements
- Programs with call statements nested within deeply-nested container statements
- Programs with single while statement procedures

Population of NextBip/NextBip* relations

NOTE: For each of the below types of SIMPLE programs, the relationships populated in the PKB was compared to the expected output. As NextBip/NextBip* relation has a natural correspondence with the CFGBip structure, its test types are identical to those of "Construction of CFGBip and Exploded CFG"

Types of test cases:

- Single procedure programs

- Multiple procedure, no call statement programs
- Procedures with call statements that have Next statements within the same procedure
- Programs with procedures that have call statements as exit statements of that procedure
- Programs with a linear call chain of procedures that have call statements as exit statements
- Programs with a call tree of procedures that have call statements as exit statements
- Programs with call statements nested within top-level container statements
- Programs with call statements nested within deeply-nested container statements
- Programs with single while statement procedures

Example of Test Case

Purpose: To test that all NextBip and NextBip* design entities can be properly extracted and stored in the PKB, while accounting for some of the edge cases as discussed above.

Required Input: A source program string representing the SIMPLE source code.

Expected Output: The PKB to be filled accordingly.

```
const std::string test_program =
    "\
procedure Bill {      \
  x = 5;              \
  call Mary;          \
  y = x + 6;          \
  call John;          \
  z = x * y + 2;      \
}                     \
                      \
procedure Mary {       \
  y = x * 3;          \
  call John;          \
  z = x + y;          \
}                     \
                      \
procedure John {       \
  if (i > 0) then {   \
    x = x + z;        \
  } else {            \
    y = x * y;        \
  }                   \
}";
```

```
source_processor::TNode root = source_processor::Parser::Parse(test_program);

WHEN( desc: "Design extractor extracts all designs") {
  PKB pkb = PKB();
  pkb.ClearAllTables();
  design_extractor::DesignExtractor::ExtractDesigns( &: pkb, root);

  THEN( desc: "PKB gets populated with NextBip relation properly") {
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(1), {2}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(2), {6}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(3), {4}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(4), {9}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(5), {}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(6), {7}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(7), {9}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(8), {3}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(9), {10, 11}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(10), {5, 8}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipStatements(11), {5, 8}));
  }

  THEN( desc: "PKB gets populated with NextBipT relation properly") {
    //easier to see by looking at wiki prof iter 2 and 3 section
    //'1. Generalized CFGBip by graph explosion'
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(1), {2, 3, 4, 5, 6, 7, 8, 9, 10, 11}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(2), {3, 4, 5, 6, 7, 8, 9, 10, 11}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(3), {4, 9, 10, 11, 5}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(4), {9, 10, 11, 5}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(5), {}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(6), {7, 9, 10, 11, 8, 3, 4, 5}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(7), {9, 10, 11, 8, 3, 4, 5}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(8), {3, 4, 9, 10, 11, 5}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(9), {10, 11, 5, 8, 3, 4, 9}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(10), {5, 8, 3, 4, 9, 10, 11}));
    REQUIRE(ContainsExactly<int>(pkb.GetNextBipTStatements(11), {5, 8, 3, 4, 9, 10, 11}));
  }
 }
}
```

Population of AffectsBip/AffectsBip* relations

NOTE: For each of the below types of SIMPLE programs, the relationships populated in the PKB was compared to the expected output

Types of test cases:

- Single procedure programs
- Multiple procedure, no call statement programs
- Procedures with call statements that call a procedure once

- Procedures with call statements that call a procedure multiple times
- Procedures with call statements from different procedures that call the same procedure
- Programs with call statements nested within top-level container statements
- Programs with call statements nested within deeply-nested container statements
- Programs with assign and read statements that may potentially “block” the path of AffectsBip as it modifies the variable in question

Example of Test Case

Purpose: To test that all AffectsBip and AffectsBip* design entities can be properly extracted and stored in the PKB, while accounting for some of the edge cases as discussed above.

Required Input: A source program string representing the SIMPLE source code.

Expected Output: The PKB to be filled accordingly.

```
const std::string test_program =
    "\
  procedure B {\
    call C; \
    while (i > 0) { \
      call C; \
    }\
    if (i > 0) then { \
      call C;\
    } else {\
      call C;\
    }\
  }\
  procedure C {\
    d = a;\
    a = b;\
    b = c;\
    c = d;\
  }";
```

```
  source_processor::TNode root = source_processor::Parser::Parse(test_program);

  WHEN( desc: "Design extractor extracts all designs") {
    PKB pkb = PKB();
    pkb.ClearAllTables();
    design_extractor::DesignExtractor::ExtractDesigns( &: pkb, root);

    THEN( desc: "PKB gets populated with AffectsBip relation properly") {
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(1), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(2), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(3), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(4), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(5), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(6), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(7), {10}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(8), {7}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(9), {8}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipStatements(10), {9}));
    }
  }
  WHEN( desc: "Design extractor extracts all designs") {
    PKB pkb = PKB();
    pkb.ClearAllTables();
    design_extractor::DesignExtractor::ExtractDesigns( &: pkb, root);

    THEN( desc: "PKB gets populated with AffectsBipT relation properly") {
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(1), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(2), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(3), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(4), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(5), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(6), {}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(7), {10, 9, 8, 7}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(8), {10, 9, 8, 7}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(9), {10, 9, 8, 7}));
      REQUIRE(ContainsExactly<int>(pkb.GetAffectedBipTStatements(10), {10, 9, 8, 7}));
    }
  }
}
```

The tests described in the test plan above should be implemented according to the schedule:

Iteration	Week	Type	Activity	Done By
3	10	Unit testing	Write tests to verify the correctness of the construction of the CFG and exploded CFG	Aaron
	11	Integration testing	Write integration tests for the population of NextBip and NextBip* in the PKB	DE-PKB (Jia Da and Zi Ying)
	12	Integration testing	Write integration tests for the population of AffectsBip and AffectsBip* in the PKB	DE-PKB (Jia Da and Zi Ying)
		System testing	Write overall Autotester test cases specifically designed to test for the extraction and evaluation of NextBip/* and AffectsBip/*	DE-PKB (Jia Da and Zi Ying) QE-PKB (Jolyn)
	13	Acceptance testing	Verify the correctness of all test cases and ensure that they run in a reasonable time frame (e.g. do not timeout)	Aaron

5.4 Design Decisions

Problem Statement: To utilize a CFGBip or Exploded Graph approach for CFGBip construction.

Constraints of the Problem:

The constraints of the problem describe conditions that all considered solutions must fulfill.

Usability to extract Extension Relations
The CFGBip is used only for the extraction of NextBip/NextBip*/AffectsBip/AffectsBip*. As such, it must contain sufficient information for the correct extraction of those relations and remain as simple as possible to prevent obscure bugs.

Evaluation Criteria:

1. Ease of construction
2. Time and memory usage
3. Ease of extracting extension relationships

Description of Possible Solutions:

CFGBip
The CFGBip is an adjacency list and is constructed similar to that of the Wiki with 2 differences. First, dummy nodes are not used. Secondly, the BranchBack edges are not labelled. BranchIn and BranchBack edges are indistinguishable from intra-procedure edges.
CFG Explosion
GENodes, a user-defined class, is used to construct the Exploded CFG. Each GENodes stores the statement number of the source program it corresponds to and a

list of pointers to GENodes it can traverse to via the NextBip relation.

Evaluation of solutions

Ease of construction	
CFGBip	CFGBip is stored as an adjacency list and 1-to-1 correspondence between nodes in the CFGBip and statement numbers of the source program is maintained, every node in the CFGBip can be accessed by indexing into a vector. This makes standard DFS traversals and creation of nodes simple.
CFG Explosion	Each node in the Exploded Graph is an object. As such, the objects themselves cannot be efficiently stored in containers. Hence, the GENodes work with pointers, which add complexity. Also, since each statement of the source program could correspond to multiple GENodes, care must be taken to ensure that the correct copy of that statement number is used whenever edges are being created.

Time and memory usage	
CFGBip	Construction of CFGBip is asymptotically similar to a DFS traversal of the CFG, and hence takes linear time with respect to the number of statements of the source program. CFGBip is stored as an adjacency list and 1-to-1 correspondence between nodes in the CFGBip and statement numbers of the source program. Further, each statement can have at most 2 NextBip statements. Hence, the number of vertices and edges is linear in the number of statements of the source program.

CFG Explosion	In the worst case, the source program contains a call hierarchy resembling a complete binary tree with a branching factor of at least 2, and would have an exponential number of nodes with respect to the number of statements of the source program. Hence, it takes at most exponential time and space to construct such an Exploded CFG.

Ease of extracting extension relationships	
CFGBip	Extracting NextBip with CFGBip is trivial as the edges directly correspond to NextBip relations. NextBip* is more involved as it requires traversal of the CFGBip while simulating a call stack. AffectsBip and AffectsBip*, however, are very complex and an algorithm which we were convinced was correct could not be devised.
CFG Explosion	Extracting NextBip with Exploded CFG is less straightforward as with CFGBip as all edges are not directly indexable, and the Exploded CFG has to be traversed like a linked list. NextBip* corresponds to a multi-source traversal, and is simpler because it does not require a call stack. AffectsBip/* can be easily extracted with the exploded graph as it requires a nested traversal of the nodes using breadth-first and depth-first traversals. With the exploded graph, the algorithm to approach extraction of AffectsBip/* became similar to the method of extracting non-interprocedural Affects/*, thus making the implementation much easier.

Final Choice:

Since our ultimate priority was correctness, and the extension relations were non-trivial, the biggest obstacle to correctness was going to be complexity which hid bugs. As such, we decided to implement both methods, with the CFGBip for the extraction of NextBip/NextBip*, and the CFG Explosion for extraction of AffectsBip/AffectsBip*. We believe that since the two CFGBip constructions best suited different extensions, the increased load to develop and test 2 CFGBip constructions is worth the decrease in algorithm complexity for extraction of the relations.

6 Coding & Documentation Standards

This section documents the coding standards used in the project, and documentation standards for API design.

6.1 Coding Standards

The coding style used by the team largely follows the convention set by the Google C++ Style Guide as seen here. Coding standards are important for many reasons. Firstly, as different developers may follow different standards or styles, deciding and adhering to a coding standard amongst a team ensures that a cohesive-looking codebase is created. This improves the code readability and extensibility as team members may be tasked to extend upon a certain component written by another member. Not only so, due to the nature of the project where multiple components are required to make API calls to each other, a standardized code style will provide for a smoother development process, where a certain level of understanding of the codebase has already been established. The styling conventions that the team has decided to adhere to for consistency are described below:

6.1.1 Header files

Almost every .cpp file has a corresponding header file (with the .h extension), which acts as the interface for other .cpp files to call and communicate with. Header files contain only the method signatures, while the respective code implementation for these methods are abstracted away in the corresponding .cpp file, making sure that a clean and implementation-free interface is provided to the developer. To improve readability and reduce clutter in the code, only header files that are absolutely necessary for the code to run will be included.

6.1.2 Namespaces

Namespaces subdivide the global scope into distinct named scopes and are useful for preventing name collisions in the global scope. As mentioned above in Section 3 Spa Design, namespaces are used to avoid polluting the global scope with classes, methods

and variables that might have the same name. Related code will be placed in the same namespace. For example, the `source_processor` namespace holds all files related to parsing and processing the source program and the `query_processor` holds all files related to parsing and processing the query program.

6.1.3 Naming conventions

Type	**Convention**	**Examples**
File name	PascalCase	`PatternClause.cpp` `DesignExtractor.cpp`
Type name (classes, structs, enums)	PascalCase	`Query Processor` `FollowsTable`
Function names	PascalCase	`InsertVariable` `GetValue`
Variable names (function parameters, data members)	snake_case	`result_type` `statement_number`

Namespace	snake_case	`source_processor`
		`design_extractor`
		`query_processor`

6.2 Documentation Standards

6.2.1 Abstract API Naming Conventions

The naming conventions in the abstract API correspond exactly to the relevant C++ classes implemented in the code. An example is given below:

Abstract API	`InsertVariable (VAR_NAME var)`
Concrete API	`InsertVariable(const std::string&)`

For this example, the PKB provides an abstract API method `InsertVariable (VAR_NAME var)` to insert a variable with the name `var` into the VarTable. Similarly, the relevant concrete API provided by the PKB has the corresponding function `InsertVariable(const std::string&)` to insert a variable into the `VarTable`. The rest of the concrete code implementation adheres to the naming conventions of the abstract API in a similar way.

6.2.2 Abstract API Types

The types used in the actual code implementation consists of native C++ types instead of the abstract API types given in Section 8.1 PKB Abstract API. For example, the abstract API utilises a type **`VAR_NAME`** to denote the variable name in the parameters of `InsertVariable (VAR_NAME var)`. However, the concrete implementation of the same function uses a string reference instead, as seen in `InsertVariable(const std::string&)`. This is consistent across all concrete implementations of the abstract API. Rather than creating a struct, native C++ types are used as there is no need to create a separate data structure that will eventually carry only one native string type. Creating a struct would add additional layers of complexity in terms of creating and accessing the data, without any additional benefits of performance.

6.2.3 Pros and Cons of an API First Approach

Using an API first approach, the team has realised that there are both benefits and downsides through the development process.

Some benefits of such an approach are:

- Using an API-first approach sets a clear plan for the developers of each component to write their own code independently, effectively decoupling the development process
- Easier to integrate as the entry points to each component are made known early in the development phase. This means that less refactoring in the later parts of the development cycle will be required
- Provides early validation of proposed APIs as the APIs are discussed amongst members before implementation, any issues can be brought up in the initial stages of planning, which means that there is less confusion and refactoring in later stages
- When APIs are finalised, this results in a single point of truth to refer to when developing, reducing the amount of discussion and improves clarity. This ensures that faster parallel development can be done, as developers can simply refer to the APIs decided during planning

However, some problems of such an approach are:

- An API first approach may delay the start of actual code implementation as more time will be spent initially drawing up the plan for the APIs
- Incomplete APIs may cause more confusion than clarity, resulting in more time being used to seek clarification before implementation
- Developers of other components may not necessarily understand how to use the APIs and will have to seek clarifications from the developer who created the API. Therefore causing a lot of back and forth discussion which might end up time consuming before code implementation can start

7 Discussion

There were no major issues in our project schedule. Everything was generally kept on track and on task via GitHub issues. Weekly meetings were also scheduled where the team would compile the lists of tasks to be done for the week and gather back at the end of the week to check in on the completion of the respective tasks. The workload would be reallocated if necessary and constant communication amongst the team made sure that the project proceeded smoothly.

There were several occurrences where the schedule was strained due to external circumstances (e.g. heavy workload in other mods), but in general work was completed on time. Additionally, team members who completed their sprints faster were able to help other team members by performing miscellaneous tasks (project management, test-script-writing) instead.

In general, project management was discovered to be quite a difficult process. The iterative breadth-first approach meant that team members were able to specialize deeply into one specific component. This had both pros and cons. It would allow team members to deeply understand and optimize their own component, but would make it harder for them to generalize into other components. Therefore, it was necessary to provide regular crash courses and updates as to the inner workings and algorithms used in each component, to ensure that all team members understood the general workings of SPA.

The importance of having a specific schedule was also learnt. Without the schedules that were planned out at the start of each iteration, it would have been very likely that the work was back-loaded. Planning out the tasks and activities that needed to be done via Github issues helped to ensure that any unforeseen problems would not make it impossible to deliver tasks on time.

8 Appendix

8.1 PKB Abstract API

1. VarTable

VarTable *Overview*: VarTable stores Variable design entity.
API
`BOOLEAN InsertVariable (VAR_NAME var);` ***Description:*** Stores the **VAR_NAME** var into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
`SET_OF_VAR_NAMES GetAllVariables();` ***Description:*** Returns a set of variable names v_1, v_2, …, v_n which are present in the SIMPLE program.

2. ProcTable

ProcTable *Overview*: ProcTable stores Procedure design entity and the range of **STMT_NO** over which it is defined.
API
`BOOLEAN InsertProcedure (PROC_NAME proc, STMT_NO start, STMT_NO`

`end);` ***Description:*** Stores the **PROC_NAME** proc with its start index and end index into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
`SET_OF_PROCEDURE_NAMES` `GetAllProcedures ();` ***Description:*** Returns a set of procedure names p_1, p_2, …, p_n which are present in the SIMPLE program.

3. Follows, Follows*

FollowsTable *Overview*: FollowsTable stores Follows design abstraction *(relationship)* between 2 design entities (stmt, stmt)
API
`BOOLEAN` `InsertFollows (`**`STMT_NO`** `stmt1,` **`STMT_NO`** `stmt2);` ***Description:*** Stores the Follows(stmt1, stmt2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
`BOOLEAN` `IsFollows (`**`STMT_NO`** `stmt_1,` **`STMT_NO`** `stmt_2);` ***Description:*** Returns TRUE if Follows(stmt_1, stmt_2) holds and the information is stored in the PKB, returns FALSE otherwise.
`STMT_NO` `GetStmtFollowedBy (`**`STMT_NO`** `stmt2);` ***Description:*** Returns a statement stmt1 which is followed by stmt2, i.e. Follows(stmt1, stmt2) holds.

	STMT_NO GetStmtFollows (**STMT_NO** stmt1); ***Description:*** Returns a statement stmt2 which follows stmt1, i.e. Follows(stmt1, stmt2) holds.
	SET_OF_STMT_NO GetAllFollowsStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which follows some other statement number.
	SET_OF_STMT_NO GetAllFollowedStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which is followed by some other statement number.

FollowsTTable *Overview*: Follows*Table stores Follows* design abstraction *(relationship)* between 2 design entities (stmt, stmt)	
	API
	BOOLEAN InsertFollowsT (**STMT_NO** stmt1, **STMT_NO** stmt2); ***Description:*** Stores the Follows*(stmt1, stmt2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN IsFollowsT (**STMT_NO** stmt_1, **STMT_NO** stmt_2); ***Description:*** Returns TRUE if Follows*(stmt_1, stmt_2) holds and the information is stored in the PKB, returns FALSE otherwise.

	SET_OF_STMT_NO GetStmtsFollowedTBy (**STMT_NO** stmt2); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which are followed by stmt2, i.e. Follows*(s_1, stmt2), Follows*(s_2, stmt2), … , Follows*(s_n, stmt2) holds.
	SET_OF_STMT_NO GetStmtsFollowsT (**STMT_NO** stmt1); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which follows stmt1, i.e. Follows*(stmt1, s_1), Follows*(stmt1, s_2), …, Follows*(stmt1, s_n) holds.
	SET_OF_STMT_NO GetAllFollowsTStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which follows* some other statement number.
	SET_OF_STMT_NO GetAllFollowedTStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which is followed* by some other statement numbers.

4. Parent, Parent*

ParentTable *Overview*: ParentTable stores Parent design abstraction *(relationship)* between 2 design entities (stmt, stmt)	
	API
	BOOLEAN InsertParent (**STMT_NO** stmt1, **STMT_NO** stmt2); ***Description:*** Stores the Parent(stmt1, stmt2) information into the PKB. Returns

	TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN IsParent (**STMT_NO** stmt1, **STMT_NO** stmt2); ***Description:*** Returns TRUE if Parent(stmt1, stmt2) holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_STMT_NO GetChildrenStatements (**STMT_NO** stmt1); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which are children statements of stmt1, i.e. Parent(stmt1, s_1), Parent(stmt1, s_2), … , Parent(stmt1, s_n) holds.
	STMT_NO GetParentStatement (**STMT_NO** stmt2); ***Description:*** Returns a statement number stmt1 which is the parent statement of stmt2, i.e. Parent(stmt1, stmt2) holds.
	SET_OF_STMT_NO GetAllParentStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which are parents to some other statement number.
	SET_OF_STMT_NO GetAllChildrenStmts(); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which are children to some other statement number.

ParentTTable

Overview: ParentStarTable stores Parent* design abstraction *(relationship)* between 2 design entities (stmt, stmt)

API

```
BOOLEAN InsertParentT (STMT_NO stmt1, STMT_NO stmt2);
```

Description: Stores the Parent*(stmt1, stmt2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.

```
BOOLEAN IsParentT (STMT_NO stmt1, STMT_NO stmt2);
```

Description: Returns TRUE if Parent*(stmt1, stmt2) holds and the information is stored in the PKB, returns FALSE otherwise.

```
SET_OF_STMT_NO GetChildrenTStatements(STMT_NO stmt1);
```

Description: Returns a list of statement numbers s_1, s_2, …, s_n which are children* of stmt1, i.e. Parent*(stmt1, s_1), Parent*(stmt1, s_2), … , Parent*(stmt1, s_n) holds.

```
SET_OF_STMT_NO GetParentTStatements(STMT_NO stmt2);
```

Description: Returns a list of statement numbers s_1, s_2, …, $s_n$which are the parent* statements of stmt2, i.e. Parent*(s_1, stmt2), Parent*(s_2, stmt2),…, Parent*(s_n, stmt2) holds.

```
SET_OF_STMT_NO GetAllParentTStmts();
```

Description: Returns a set of all statement numbers s_1, s_2, …, s_n which are parent*

	to some other statement numbers.
	SET_OF_STMT_NO GetAllChildrenTStmts(); ***Description:*** Returns a set of all statement numbers s_1, s_2, …, s_n which are children of some other statement numbers.

5. Modifies

ModifiesTable *Overview*: ModifiesTable stores Modifies design abstraction *(relationship)* between 2 design entities (stmt, variable) or (procedure, variable)	
	API
	BOOLEAN InsertModifies (**STMT_NO** stmt, **VAR_NAME** var); ***Description:*** Stores the Modifies(stmt, var) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN IsModifies (**STMT_NO** stmt, **VAR_NAME** var); ***Description:*** Returns TRUE if Modifies(stmt, var) holds and the information is stored in the PKB, returns FALSE otherwise.
	BOOLEAN InsertModifies (**PROC_NAME** proc, **VAR_NAME** var); ***Description:*** Stores the Modifies(proc, var) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in

	the PKB.
	BOOLEAN IsModifies (**PROC_NAME** proc, **VAR_NAME** var); ***Description:*** Returns TRUE if Modifies(proc, var) holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_VARIABLES GetModifiedVariables (**STMT_NO** stmt); ***Description:*** Returns a set of variables v_1, v_2, …, v_n which are modified by statement stmt, i.e. Modifies(stmt, v_1), Modifies(stmt, v_2), …, Modifies(stmt, v_n) holds.
	SET_OF_STMT_NO GetModifiesStatements (**VAR_NAME** var); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which modifies a variable var, i.e Modifies(s_1, var), Modifies(s_2, var), …, Modifies(s_n, var) holds.
	SET_OF_VARIABLES GetModifiedVariables (**PROC_NAME** proc); ***Description:*** Returns a set of variables v_1, v_2, …, v_n which are modified by statement stmt, i.e. Modifies(stmt, v_1), Modifies(stmt, v_2), …, Modifies(stmt, v_n) holds.
	SET_OF_PROC_NAME GetModifiesProcedures (**VAR_NAME** var); ***Description:*** Returns a set of procedure names p_1, p_2, …, p_n which modifies a variable var, i.e Modifies(p_1, var), Modifies(p_2, var), …, Modifies(p_n, var) holds.
	SET_OF_STMT_NO GetAllModifiesStatements ();

	Description: Returns a unique set of all statements which holds Modifies relationships.
	`SET_OF_PROC_NAME GetAllModifiesProcedures ();` ***Description:*** Returns a unique set of all procedures which holds Modifies relationships.

6. Uses

UsesTable *Overview*: UsesTable stores Uses design abstraction *(relationship)* between 2 design entities (stmt, variable) or (procedure, variable)	
	API
	`BOOLEAN InsertUses (STMT_NO stmt, VAR_NAME var);` ***Description:*** Stores the Uses(stmt, var) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	`BOOLEAN IsUses (STMT_NO stmt, VAR_NAME var);` ***Description:*** Returns TRUE if Uses(stmt, var) holds and the information is stored in the PKB, returns FALSE otherwise.
	`BOOLEAN InsertUses(PROC_NAME proc, VAR_NAME var);` ***Description:*** Stores the Uses(proc, var) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.

	BOOLEAN IsUses (**PROC_NAME** proc, **VAR_NAME** var); ***Description:*** Returns TRUE if Uses(proc, var) holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_VARIABLES GetUsedVariables (**STMT_NO** stmt); ***Description:*** Returns a set of variables v_1, v_2, …, v_n which are used by statement stmt, i.e. Uses(stmt, v_1), Uses(stmt, v_2), …, Uses(stmt, v_n) holds.
	SET_OF_STMT_NO GetUsesStatements (**VAR_NAME** var); ***Description:*** Returns a set of statement numbers s_1, s_2, …, s_n which uses a variable var, i.e Uses(s_1, var), Uses(s_2, var), …, Uses(s_n, var) holds.
	SET_OF_VARIABLES GetUsedVariables (**PROC_NAME** proc); ***Description:*** Returns a set of variables v_1, v_2, …, v_n which are modified by statement stmt, i.e. Uses(stmt, v_1), Uses(stmt, v_2), …, Uses(stmt, v_n) holds.
	SET_OF_PROC_NAME GetUsesProcedures (**VAR_NAME** var); ***Description:*** Returns a set of procedure names p_1, p_2, …, p_n which uses a variable var, i.e Uses(p_1, var), Uses(p_2, var), …, Uses(p_n, var) holds.
	SET_OF_STMT_NO GetAllUsesStatements (); ***Description:*** Returns a unique set of all statements which holds Uses relationship.
	SET_OF_PROC_NAME GetAllUsesProcedures ();

Description: Returns a unique set of all procedures which holds Uses relationship.

General Design Entities *Overview*: These are Insertion APIs which populate information about specific design entities
API
BOOLEAN InsertVariable(**VAR_NAME** var); ***Description:*** Stores the variable identifier var into the PKB if it appears in the SIMPLE source program. Returns TRUE if information is added successfully or False if information already exists in the PKB.
BOOLEAN InsertStatement(**STMT_NO** stmt); ***Description:*** Stores the statement number into the PKB if it appears in the SIMPLE source program. Returns TRUE if information is added successfully or False if information already exists in the PKB.
BOOLEAN InsertConstant(**CONST_VAL** val); ***Description:*** Stores the value val into the PKB if it is a constant in the SIMPLE source program. Returns TRUE if information is added successfully or False if information already exists in the PKB.
BOOLEAN InsertIf(**STMT_NO** stmt, **LIST_OF_VARIABLES** variables); ***Description:*** Stores the statement number and the variables used in the condition

	into the PKB if it is an “IF” statement in the SIMPLE source program. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN InsertWhile(**STMT_NO** stmt, **LIST_OF_VARIABLES** variables); ***Description:*** Stores the statement number and the variables used in the condition into the PKB if it is a “WHILE” statement in the SIMPLE source program. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN InsertRead(**STMT_NO** stmt, **VAR_NAME** var); ***Description:*** Stores read statement info into the PKB if stmt is a “READ” statement and reads into var. Returns TRUE if information is added successfully or False if information already exists in the PKB
	BOOLEAN InsertPrint(**STMT_NO** stmt, **VAR_NAME** var); ***Description:*** Stores print statement info into the PKB if stmt is a “PRINT” statement and prints value of var. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN InsertAssignment(**STMT_NO** stmt, **VAR_NAME** lhs, **TOKEN_LIST** rhs); ***Description:*** Stores the assignment info into the PKB if stmt is an “ASSIGN” statement with lhs being assigned to, and rhs is the assigned expression in reverse polish notation. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN InsertProcedure(**VAR_NAME** var, **STMT_NO** start, **STMT_NO** end); ***Description:*** Stores procedure info into the PKB if procedure var exists and has

	start and end as the lowest and highest statement numbers directly nested within it respectively. Returns TRUE if information is added successfully or False if information already exists in the PKB.

7. Calls/Calls*

CallsTable *Overview*: CallsTable stores Calls design abstraction *(relationship)* between 2 procedures `(proc1, proc2)`	
	API
	`BOOLEAN` `InsertCalls (`**`PROC_NAME`** `proc1,` **`PROC_NAME`** `proc2);` ***Description:*** Stores the `Calls(proc1, proc2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	`BOOLEAN` `IsCalls (`**`PROC_NAME`** `proc1,` **`PROC_NAME`** `proc2);` ***Description:*** Returns TRUE if `Calls(proc1, proc2)` holds and the information is stored in the PKB, returns FALSE otherwise.
	`SET_OF_PROC` `GetProceduresThatCalls(`**`PROC_NAME`** `proc);` ***Description:*** Returns a set of procedures which `Calls` a given procedure `proc`.
	`SET_OF_PROC` `GetProceduresCalled (`**`PROC_NAME`** `proc);` ***Description:*** Returns a set of procedures which are Called by a given procedure `proc`.
	`SET_OF_PROC` `GetAllCallsProcedures();` ***Description:*** Returns a set of all procedures which `Calls` one or more other

	procedures.
	SET_OF_PROC `GetAllCalledProcedures();` ***Description:*** Returns a set of procedures which are Called by one or more other procedures.

CallsTTable *Overview*: CallsTTable stores CallsT design abstraction *(relationship)* between 2 procedures `(proc1, proc2)`	
	API
	BOOLEAN `InsertCallsT (`**PROC_NAME** `proc1,` **PROC_NAME** `proc2);` ***Description:*** Stores the `CallsT(proc1, proc2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN `IsCallsT (`**PROC_NAME** `proc1,` **PROC_NAME** `proc2);` ***Description:*** Returns TRUE if `CallsT(proc1, proc2)` holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_PROC `GetProceduresThatCallsT (`**PROC_NAME** `proc);` ***Description:*** Returns a set of procedures which `CallsT` a given procedure `proc`.
	SET_OF_PROC `GetProceduresCalledT (`**PROC_NAME** `proc);` ***Description:*** Returns a set of procedures which are CalledT by a given procedure `proc`.
	SET_OF_PROC `GetAllCallsTProcedures();` ***Description:*** Returns a set of all procedures which `CallsT` one or more other

	procedures.
	SET_OF_PROC GetAllCalledTProcedures(); ***Description:*** Returns a set of procedures which are CalledT by one or more other procedures.

8. Next/Next*

NextTable *Overview*: NextTable stores Next design abstraction *(relationship)* between 2 prog_lines (prog_line1, prog_line2)	
	API
	BOOLEAN InsertNext (**PROC_LINE** prog_line1, **PROG_LINE** prog_line2); ***Description:*** Stores the Next(prog_line1, prog_line2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN IsNext (**PROG_LINE** prog_line1, **PROC_LINE** prog_line2); ***Description:*** Returns TRUE if Next(prog_line1, prog_line2) holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_PROC GetNextStatements(**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which is Next to a given program line prog_line.
	SET_OF_PROC GetPreviousStatements (**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which are Previous to a given program line prog_line.

SET_OF_PROC GetAllNextStatements(); ***Description:*** Returns a set of all program lines which are Next to one or more other program lines.
SET_OF_PROC GetAllPreviousStatements(); ***Description:*** Returns a set of all program lines which are Previous to one or more other program lines.

NextTTable *Overview*: NextTTable stores NextT design abstraction *(relationship)* between 2 prog_lines (prog_line1, prog_line2)
API
BOOLEAN InsertNextT (**PROC_LINE** prog_line1, **PROG_LINE** prog_line2); ***Description:*** Stores the NextT(prog_line1, prog_line2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
BOOLEAN IsNextT (**PROG_LINE** prog_line1, **PROC_LINE** prog_line2); ***Description:*** Returns TRUE if NextT(prog_line1, prog_line2) holds and the information is stored in the PKB, returns FALSE otherwise.
SET_OF_PROC GetNextTStatements(**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which is NextT to a given program line prog_line.
SET_OF_PROC GetPreviousTStatements (**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which are PreviousT to a given program line prog_line.

	SET_OF_PROC `GetAllNextTStatements();` ***Description:*** Returns a set of all program lines which are `NextT` to one or more other program lines.
	SET_OF_PROC `GetAllPreviousTStatements();` ***Description:*** Returns a set of all program lines which are PreviousT to one or more other program lines.

9. Affects/Affects*

AffectsTable *Overview*: AffectsTable stores Affects design abstraction *(relationship)* between 2 assign statements `(stmt1, stmt2)`	
	API
	BOOLEAN `InsertAffects (`**STMT_NO** `stmt1,` **STMT_NO** `stmt2);` ***Description:*** Stores the `Affects(stmt1, stmt2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN `IsAffects (`**STMT_NO** `stmt1,` **STMT_NO** `stmt2);` ***Description:*** Returns TRUE if `Affects(stmt1, stmt2)` holds and the information is stored in the PKB, returns FALSE otherwise.
	SET_OF_PROC `GetStatementsThatAffects(`**STMT_NO** `stmt2);` ***Description:*** Returns a set of statements which `Affects` a given statement `stmt2`.
	SET_OF_PROC `GetAffectedStatements (`**STMT_NO** `stmt1);` ***Description:*** Returns a set of statements which are affected by a given program

	line `stmt1`.
	`SET_OF_PROC` `GetAllAffectsStatements();` ***Description:*** Returns a set of all statements which `Affects` one or more other statements.
	`SET_OF_PROC` `GetAllAffectedStatements();` ***Description:*** Returns a set of all statements which are Affected by one or more other statements.

AffectsTTable *Overview*: AffectsTTable stores AffectsT design abstraction *(relationship)* between 2 assign statements `(stmt1, stmt2)`	
	API
	`BOOLEAN` `InsertAffectsT (`**`STMT_NO`** `stmt1,` **`STMT_NO`** `stmt2);` ***Description:*** Stores the `AffectsT(stmt1, stmt2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	`BOOLEAN` `IsAffectsT (`**`STMT_NO`** `stmt1,` **`STMT_NO`** `stmt2);` ***Description:*** Returns TRUE if `AffectsT(stmt1, stmt2)` holds and the information is stored in the PKB, returns FALSE otherwise.
	`SET_OF_PROC` `GetStatementsThatAffectsT(`**`STMT_NO`** `stmt2);` ***Description:*** Returns a set of statements which `AffectsT` a given statement `stmt2`.
	`SET_OF_PROC` `GetAffectedTStatements (`**`STMT_NO`** `stmt1);`

	Description: Returns a set of statements which are AffectedT by a given program line `stmt1`.
	`SET_OF_PROC` `GetAllAffectsTStatements();` ***Description:*** Returns a set of all statements which `AffectsT` one or more other statements.
	`SET_OF_PROC` `GetAllAffectedTStatements();` ***Description:*** Returns a set of all statements which are AffectedT by one or more other statements.

10. NextBip/NextBip*

NextBipTable *Overview*: NextBipTable stores NextBip design abstraction *(relationship)* between 2 prog_lines `(prog_line1, prog_line2)`	
	API
	`BOOLEAN` `InsertNextBip (`**`PROC_LINE`** `prog_line1,` **`PROG_LINE`** `prog_line2);` ***Description:*** Stores the `NextBip(prog_line1, prog_line2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	`BOOLEAN` `IsNextBip (`**`PROG_LINE`** `prog_line1,` **`PROC_LINE`** `prog_line2);` ***Description:*** Returns TRUE if `NextBip(prog_line1, prog_line2)` holds and the information is stored in the PKB, returns FALSE otherwise.
	`SET_OF_PROC` `GetNextBipStatements(`**`PROG_LINE`** `prog_line);` ***Description:*** Returns a set of program lines which is `NextBip` to a given program line `prog_line`.

SET_OF_PROC GetPreviousBipStatements (**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which are PreviousBip to a given program line prog_line.
SET_OF_PROC GetAllNextBipStatements(); ***Description:*** Returns a set of all program lines which are NextBip to one or more other program lines.
SET_OF_PROC GetAllPreviousBipStatements(); ***Description:*** Returns a set of all program lines which are PreviousBip to one or more other program lines.

NextBipTTable *Overview*: NextBipTTable stores NextBipT design abstraction *(relationship)* between 2 prog_lines (prog_line1, prog_line2)
API
BOOLEAN InsertNextBipT (**PROC_LINE** prog_line1, **PROG_LINE** prog_line2); ***Description:*** Stores the NextBipT(prog_line1, prog_line2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
BOOLEAN IsNextBipT (**PROG_LINE** prog_line1, **PROC_LINE** prog_line2); ***Description:*** Returns TRUE if NextBipT(prog_line1, prog_line2) holds and the information is stored in the PKB, returns FALSE otherwise.
SET_OF_PROC GetNextBipTStatements(**PROG_LINE** prog_line); ***Description:*** Returns a set of program lines which is NextBipT to a given program

	line `prog_line`.
	`SET_OF_PROC` `GetPreviousBipTStatements (`**`PROG_LINE`** `prog_line);` ***Description:*** Returns a set of program lines which are PreviousBipT to a given program line `prog_line`.
	`SET_OF_PROC` `GetAllNextBipTStatements();` ***Description:*** Returns a set of all program lines which are `NextBipT` to one or more other program lines.
	`SET_OF_PROC` `GetAllPreviousBipTStatements();` ***Description:*** Returns a set of all program lines which are PreviousBipT to one or more other program lines.

11. AffectsBip/AffectsBip*

AffectsBipTable *Overview*: AffectsBipTable stores AffectsBip design abstraction *(relationship)* between 2 assign statements `(stmt1, stmt2)`	
	API
	`BOOLEAN` `InsertAffectsBip (`**`STMT_NO`** `stmt1,` **`STMT_NO`** `stmt2);` ***Description:*** Stores the `AffectsBip(stmt1, stmt2)` information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	`BOOLEAN` `IsAffectsBip (`**`STMT_NO`** `stmt1,` **`STMT_NO`** `stmt2);` ***Description:*** Returns TRUE if `AffectsBip(stmt1, stmt2)` holds and the information is stored in the PKB, returns FALSE otherwise.

	SET_OF_PROC GetStatementsThatAffectsBip(**STMT_NO** stmt2); ***Description:*** Returns a set of statements which AffectsBip a given statement stmt2.
	SET_OF_PROC GetAffectedBipStatements (**STMT_NO** stmt1); ***Description:*** Returns a set of statements which are AffectedBip by a given program line stmt1.
	SET_OF_PROC GetAllAffectsBipStatements(); ***Description:*** Returns a set of all statements which AffectsBip one or more other statements.
	SET_OF_PROC GetAllAffectedBipStatements(); ***Description:*** Returns a set of all statements which are AffectedBip by one or more other statements.

AffectsBipTTable *Overview*: AffectsBipTTable stores AffectsBipT design abstraction *(relationship)* between 2 assign statements (stmt1, stmt2)	
	API
	BOOLEAN InsertAffectsBipT (**STMT_NO** stmt1, **STMT_NO** stmt2); ***Description:*** Stores the AffectsBipT(stmt1, stmt2) information into the PKB. Returns TRUE if information is added successfully or False if information already exists in the PKB.
	BOOLEAN IsAffectsBipT (**STMT_NO** stmt1, **STMT_NO** stmt2); ***Description:*** Returns TRUE if AffectsBipT(stmt1, stmt2) holds and the information is stored in the PKB, returns FALSE otherwise.

	SET_OF_PROC GetStatementsThatAffectsBipT(**STMT_NO** stmt2); ***Description:*** Returns a set of statements which AffectsBipT a given statement stmt2.
	SET_OF_PROC GetAffectedTStatements (**STMT_NO** stmt1); ***Description:*** Returns a set of statements which are AffectedBipT by a given program line stmt1.
	SET_OF_PROC GetAllAffectsBipTStatements(); ***Description:*** Returns a set of all statements which AffectsBipT one or more other statements.
	SET_OF_PROC GetAllAffectedBipTStatements(); ***Description:*** Returns a set of all statements which are AffectedBipT by one or more other statements.

8.2 Query Optimization Data

8.2.1 Sorting of Design Abstractions

Refer to Section 3.5.5.5 for the context of this section.

To determine the average result space of each Design Abstraction, the team's random source code generator was used with varied inputs to generate 40 random source codes. The following query was then used to extract the result space of all 16 design abstractions for each source code:

```
stmt s1, s2, s3, s4, s5, s6, s7, s8, s9, s10; prog_line n1, n2, n3, n4, n5, n6, n7, n8; assign a1, a2, a3, a4, a5, a6, a7, a8; procedure p1, p2, p3, p4; variable v1, v2;

Select BOOLEAN such that Follows(s1, s2) and Follows*(s3, s4) and Parent(s5, s6) and Parent*(s7, s8) and Uses(s9, v1) and Modifies(s10, v2) and Next(n1, n2) and Next*(n3, n4) and NextBip(n5, n6) and NextBip*(n7, n8) and Affects(a1, a2) and Affects*(a3, a4) and AffectsBip(a5, a6) and AffectsBip*(a7, a8) and Calls(p1, p2) and Calls*(p3, p4)
```

The average size (number of rows) of the tables for each design abstraction was then calculated and the results are as follows:

```
Calls: 4.08
Calls*: 5.6
Affects: 58.27
Follows: 157.55
AffectsBip: 186.93
Parent: 190.15
Affects*: 205.03
Next: 268.23
```

```
NextBip: 277.62
Follows*: 498.15
Parent*: 503.1
Modifies: 854.27
Uses: 1713.53
AffectsBip*: 1773.38
Next*: 16601.92
NextBip*: 57966.95
```

8.2.2 Query Evaluation Time for Different Optimizations

To verify the effectiveness of the 5 different query optimizations that were attempted, two autotester test cases were utilised. One test case included a smaller source code of ~150 lines and some complex, multiple clause queries, and the other test case was adversarial and meant to target inefficient evaluation. Since most queries in the adversarial test case will break or TLE without any optimisations, optimisations will be toggled off one at a time to compare results. For the simpler test case, optimisations will be turned on incrementally.

In the below test data, greyed boxes are used as a benchmark, and a green box signifies that the said optimization did in fact improve overall performance (degree of such is based on the shade of green) and a red box indicates a regression.

Complex queries test case

For this test case, the autotester will run 10 times for each optimization added, and the average total time taken by the **Query Evaluator only** will be divided by the number of test cases to obtain the average time taken for each query.

Optimization	Average time taken for each query (ms)
All optimizations turned off	37.956
Remove duplicates	38.655
Remove duplicates + Sort Clauses	41.735
Remove duplicates + Sort Clauses + Group Before Merge	0.589
Remove duplicates + Sort Clauses + Group Before Merge + Sort Before BFS	0.437
Remove duplicates + Sort Clauses + Group Before Merge + Sort Before BFS + Sort Neighbours	0.424

Adversarial Test Case

Since each of these test cases are likely to tackle one or two particular optimizations, it might be useful to see the performance of each query separately. The test case will run 5 times for each optimization, and the average time is taken. For queries that cannot be evaluated in an acceptable time (>100000ms), a TLE will be recorded. Similar to the queries from the previous test case, the time recorded excludes the time taken for the Query Parser.

Query	All Opt	Without Remove Duplicate	Without Sort Clauses	Without Grouping & relevant optimizations	Without Sort Before BFS	Without Sort Neighbours
1	0.564	0.707	0.513	0.460	0.586	0.483

2	0.391	32.127	0.387	0.430	0.391	0.376
3	0.108	1.026	0.082	0.079	0.094	0.079
4	0.013	0.013	134.870	0.013	0.013	0.012
5	117.246	122.336	69.705	70490.73	149.093	118.014
6	87.266	88.576	90.359	TLE	88.070	83.686
7	29.322	30.406	48.651	65.560	30.190	28.498
8	92.183	94.949	91.044	TLE	94.741	88.809
9	0.216	0.206	80.011	0.223	0.239	0.217
10	2.124	2.123	2.076	330.212	2.117	1.971
11	0.223	0.214	0.240	0.258	0.216	0.225
12	200.332	212.071	225.644	254.492	210.419	197.986
13	22.175	23.953	22.972	23.577	24.239	33.741
14	45.052	46.576	96.230	587.750	47.391	43.240
15	151.078	153.607	153.074	TLE	156.498	150.350
16	80.809	81.355	164.612	54957.510	83.102	81.297
17	363.579	358.420	362.557	511.963	360.469	361.006
Avg	66.265	69.374	85.728	8481.555	69.330	66.124
Avg < 5000ms	66.265	69.374	85.728	147.896	69.330	66.124