13 November, 2014

CS3202 Software Engineering Project

Team Number: 4 Consultation Day/Hour: Tuesday, 1pm

Team Name: Team 4

Team Members Information:

Group PKB

Saloni Kaur	A0084053L	a0084053@nus.edu.sg
M I Azima	A0085594N	a0085594@nus.edu.sg

Group PQL

Saima Mahmood	A0084176Y	a0084176@nus.edu.sg
Nguyen Trong Son	A0088441B	A0088441@nus.edu.sg
Vu Phuc Tho	A0090585X	A0090585@nus.edu.sg

Contents

0. Project Story

CS3202 has been an eventful project based module. In the continuation from CS3201, we have implemented the full scope of the Static Program Analyzer (SPA). We ensured that we improved on the weaknesses from our previous project and strengthened the base to build our SPA upon. We also had the addition of a new member, alongside one member of our group opting to change groups. This switch of members actually improved the dynamics of our group, allowing us to work better with one another. The remainder of this report shall discuss in detail how we went about implementing the various components of the SPA. This discussion shall include a summary of our main achievements (Section 1), project plans (Section 2), UML diagrams (Section 3), design decisions (section 4), coding standards and experiences (Section 5), query processing (Section 6), testing (Section 7) and finally end off with a concluding discussion (Section 8).

1. Summary of Main Achievements

1.1. Basic SPA Functionality

For the purposes of the CS3202 development of the SPA, we have implemented the full SPA as described in the Project Handbook. This includes the implementation of the components:

- Parser
- Design Extractor
- Program Knowledge Base (PKB)
- Query Processor (QP).

The PKB stores the design abstractions implemented:

- Abstract Syntax Tree (AST)
- Follows/Follows*
- Parent/Parent*
- Modifies
- Uses
- Calls/Calls*
- Next/Next*
- Affects/Affects*

The QP handles the processing of queries involving the aforementioned design abstractions alongside a combination of “with”, “such that” and “pattern” clauses. It has been implemented to also return tuple results. The QP also includes components that handle the optimization of query evaluation.

The description above just highlights the main functionality implemented. Overall, all of the required functions from iteration 1-3 have been implemented. The details of their implementation shall be discussed in the later sections of this report.

1.2. Highlights of System

In the implementation of the functionality, defined by the handbook, we have ensured that aspects of our software standout from the norm. This is in the way that we have implemented some functions and also in the addition of certain components. The main highlights of our project includes:

- Polymorphism of data structure for PKB (`MapTable` & `ListTable`)
- Next* Implementation
- Addition of Query Representator (QR) and Query Optimizer (QO) in the QP
- Query Evaluator (QE) in the QP for managing temporary results

These highlights will be further elaborated on in Section 4 and Section 6.

1.3. Extension for Bonus Points

We had intended to implement the first extension for the extended code pattern. Which includes the relationships Contains, Contains* and Siblings. We had managed to implement the PKB part of this extension, however due to a lack of time we were not able to integrate this with the QP.

2. Project Plans

2.1. Project Schedule

The tables in this section show how we distributed the work into various tasks throughout the 4 main iterations.

2.1.1. Iteration 1

Team Member	Testing	Writing Test Cases	Revamp of PKB Tables	Refractoring QP	Working on QP	Extending Parser Functionality
Azima	*	*	*			
Saima	*				*	
Saloni	*		*			
Sean	*					*
Tho	*			*	*	

Table 2.1: Iteration 1 Work Distribution

Since this was the starting iteration, we had mainly focused on revamping the design of the whole system, from the Revision of the Prototype iteration. As a result, we were unable to finish implementing all of the required functionality defined in iteration 1.

2.1.2. Iteration 2

Team Member	Testing	Writing Test Cases	Revamp of TNODE	Next for PKB	Working on QP	Fixing issues with Parser	Report
Azima	*	*					
Saima	*	*			*		*
Saloni	*	*		*			
Sean	*					*	
Tho	*		*		*		*

Table 2.2: Iteration 2 Work Distribution

This iteration saw us complete all of the requirements from iteration 1 and iteration 2. The main setback in this iteration was in the design of the `TNode` data structure. This class is used to build various trees used for data storage. Previously we had implemented the `TNode` without the use of pointers. This means that every time we would pass data, the `TNode` would create a new copy instead of passing the original data from one function to another. This would slow down the SPA process, since each time we would add a new node, the existing tree would be copied another time to attach to the new node. This was seen to be a waste of processing time and storage. It slowed our system down considerably.

We overcame the aforementioned problem by creating a new `TNode` and assigning a pointer to its address. Basically, creating a pointer of type `TNode`.

Old way:

```
TNode node();

TNode * pointer = &node;
```

New way:

```
TNode * pointer = new TNode();
```

2.1.3. Iteration 3

Team Member	Testing	Writing Test Cases	Revamp of Relationships Data Structure	Affects	Working on QP	Fixing Issues with QE	Report
Azima	*	*					*
Saima	*				*		*
Saloni	*	*					*
Sean	*		*	*			
Tho	*				*	*	

Table 2.3: Iteration 3 Work Distribution

Iteration 3 saw us having a number of issues with the implementation of the Next * and Affects relationships. Also we had to improve on the managing of temporary results in the Query Evaluator, for multiple clauses. What these issues were and how we fixed them would be discussed in Section 4 and 6 respectively. This slowed us down considerably in this iteration and we could carry out aggressive testing, as we had initially planned to.

2.1.4. Iteration 4

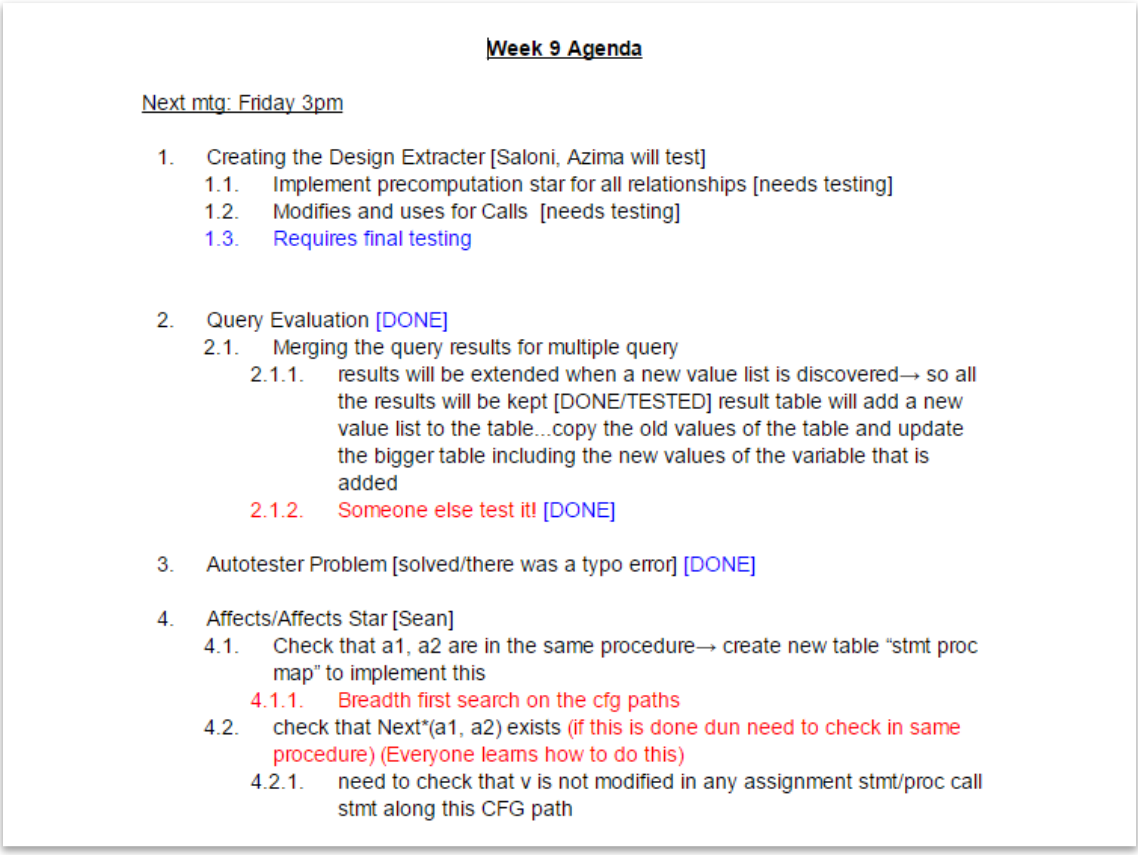
Team Member	Testing	Writing Test Cases	Extension Relationships	Fixing Issues with System	Report
Azima	*	*		*	*
Saima	*	*	*	*	*
Saloni	*	*	*	*	*
Sean	*			*	*
Tho	*			*	*

Table 2.4: Iteration 4 Work Distribution

The emphasis placed on this iteration was to carry out vigorous testing on the system and implement the extension relations, Siblings and Contains, time permitting.

2.2. Organization of Meetings

As this project was implemented in a total of 4 iterations, we measured our progress based on the state of our SPA in the previous iteration and the requirements for the current iteration. To make sure that we stayed consistent we met, on average, at least 3 times a week, including every Tuesday which was our consultation time slot. In the first meeting of the week we would set an agenda for the things to be achieved in that specific meeting, and for that week. The subsequent meetings would be organized around our tutor's feedback and on improving our system based on the critique we would have received. Fig 2.1 below shows an excerpt from one of our meeting's agenda.



Week 9 Agenda

Next mtg: Friday 3pm

1. Creating the Design Extracter [Saloni, Azima will test]
   - 1.1. Implement precomputation star for all relationships [needs testing]
   - 1.2. Modifies and uses for Calls [needs testing]
   - 1.3. Requires final testing

2. Query Evaluation [DONE]
   - 2.1. Merging the query results for multiple query
     - 2.1.1. results will be extended when a new value list is discovered→ so all the results will be kept [DONE/TESTED] result table will add a new value list to the table...copy the old values of the table and update the bigger table including the new values of the variable that is added
     - 2.1.2. Someone else test it! [DONE]

3. Autotester Problem [solved/there was a typo error] [DONE]

4. Affects/Affects Star [Sean]
   - 4.1. Check that a1, a2 are in the same procedure→ create new table "stmt proc map" to implement this
     - 4.1.1. Breadth first search on the cfg paths
   - 4.2. check that Next*(a1, a2) exists (if this is done dun need to check in same procedure) (Everyone learns how to do this)
     - 4.2.1. need to check that v is not modified in any assignment stmt/proc call stmt along this CFG path

Figure 2.1: Agenda Excerpt

3. UML Diagrams

The UML sequence diagrams presented in this section display how the SPA program flow works between the Parser, PKB and QP. These diagrams allowed us to visualize the various component interaction of the SPA and thus aided in the project planning.

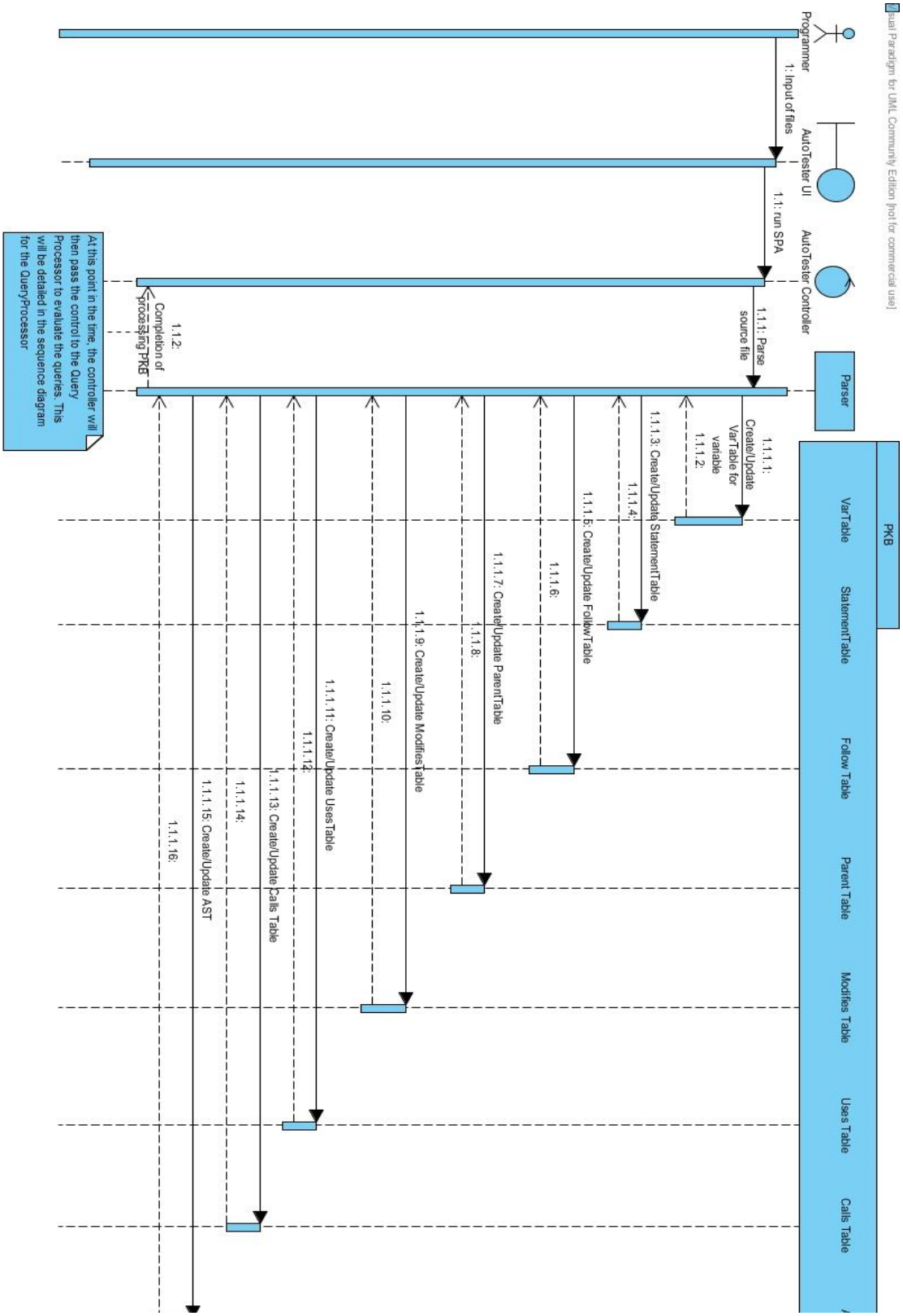


Figure 3.1: Sequence Diagram for Processing PKB

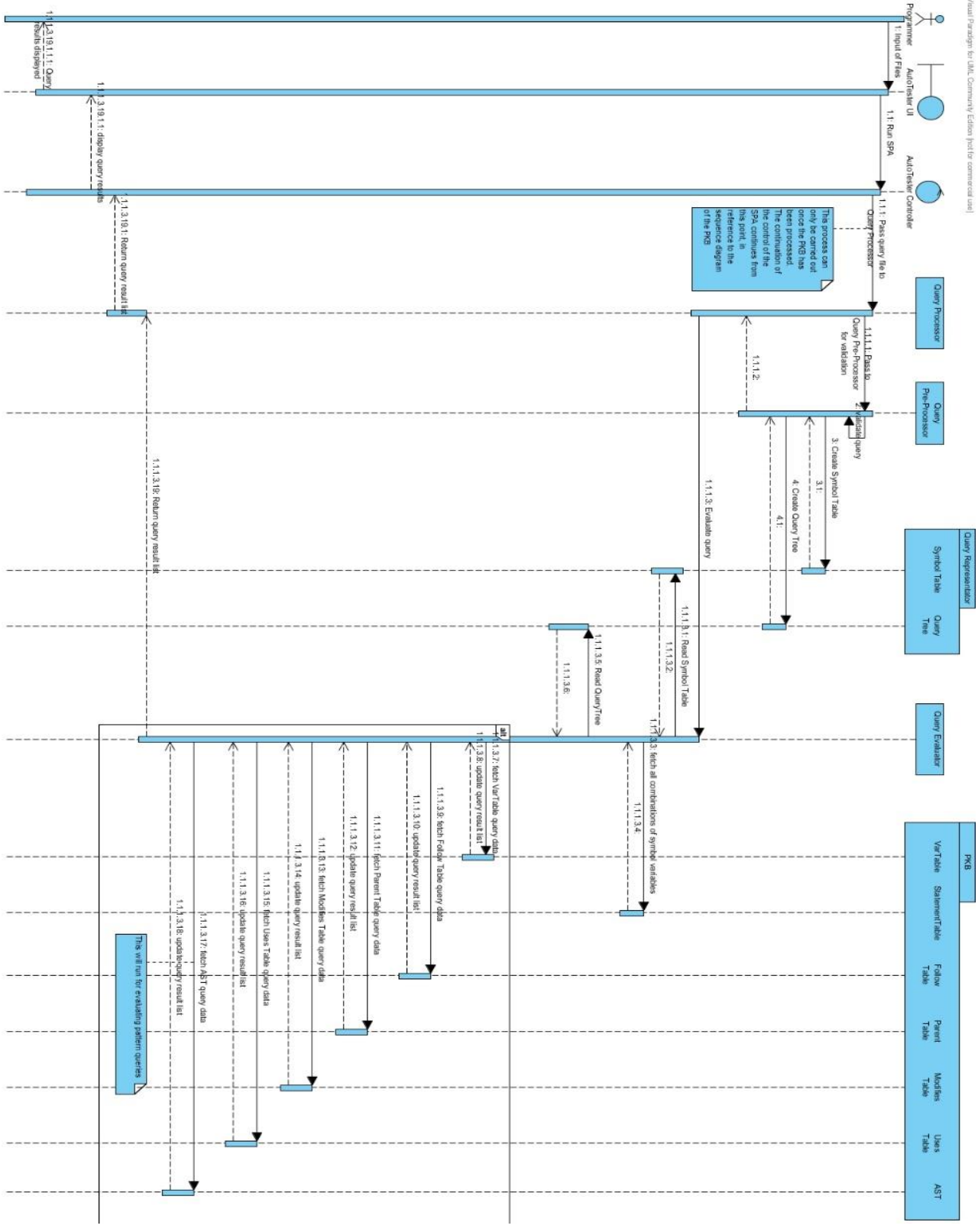


Figure 3.2: Sequence Diagram for Query Processor Flow

These 2 diagrams shown in Fig 3.1 and Fig 3.2, allowed us to understand the system properly before we went to code it out. In making these UML diagrams, we are able to separate the various components and abstract out the most important things. This enabled us to look at each decomposed aspect of the system and come up with an in-depth system architecture. This in turn allowed us to see what the basic requirements for each component were.

4. Design Decisions

From the Handbook we note that the SPA should have the following program quality attributes:

- Reliability of Programs
- Flexibility of SPA
- Reusability, Scalability
- Performance of Query Evaluation

Keeping all of these requirements in mind we have ensured that our SPA follows the principles of good design decisions. The main aim of these design decisions is to ensure that there is low coupling and high cohesion.

As CS3202 is a continuation module, we had ensured that, for the most, we started off with standardizing the way we code. This included naming conventions, which will be described in section 5. Also we had a rough idea of the high-level breakdown of the system. The figure below shows a part of the system architecture in which we added our own components (differs from handbook), more specifically that of the QP.

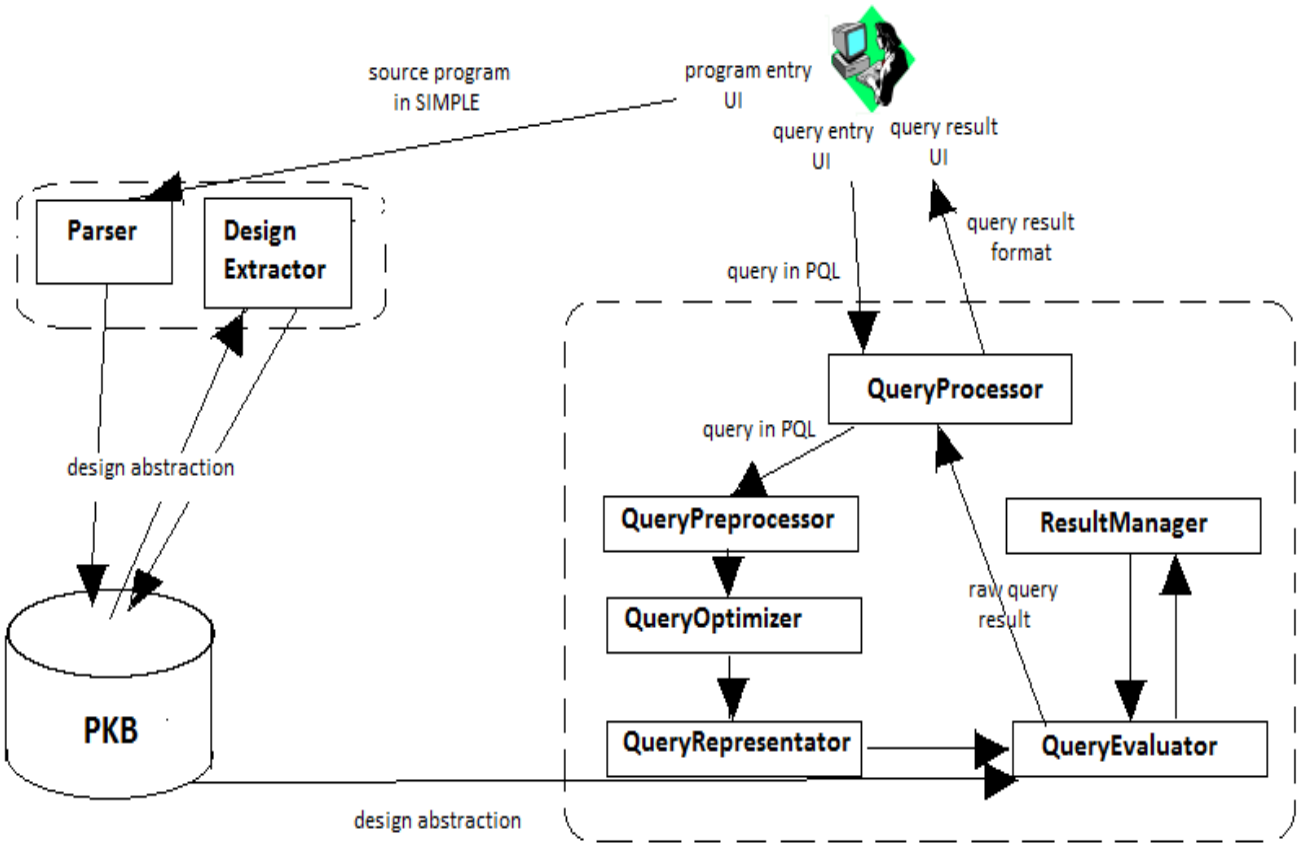


Figure 4.1: System Architecture Diagram for SPA

We have implemented our SPA based on this system architecture. The organization of the system in this manner ensures that each component only has one specific task assigned to it. The assignment of tasks become more and more specific as we go down to the lower levels of implementation. For instance looking at the component of the QP, we see how the addition of the 2 extra sub-components, QR and QO, ensures that there is a greater separation of concern and in turn higher information hiding. This is an example of how we have ensured that our SPA reduces the coupling between components and in turn increase cohesion.

4.1. PKB Data Structure Representation

The most import highlight of our system is how we went about implementing the data structure for the tables in the PKB. This section details the process through which we came up with the best way to store all of the relationships.

Initial Implementation

In the beginning we had implemented the PKB using 2 common ways, via a table and a 2D vector. The diagram below illustrates the method used and their respective pros and cons.

Table

	4	5	6
4	0	1	0
5	0	0	1
6	0	0	0

Pros:

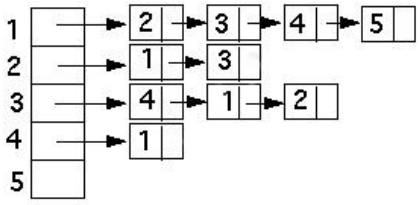
- Boolean checking for relationships, eg `isParent(4,6)`, can be done in O(1).

Cons:

-Retrieving all data relating to a key, such as get methods getParent(s), will be O(N).

-When parsing the source code, we cannot set an initial table size and have to resize every time the contained data reaches the limit. This resize process has the complexity of $O(N^2)$.

2D Vectors



Pros:

-Retrieving all data relating to a key, eg `getParent(4)`, can be done in O(1).

Cons:

-Boolean Relationship checking, eg `isParent(4,6)`, takes O(N).

Figure 4.2: Previous Implementation of PKB Tables

Given the pros and cons of the 2 aforementioned implementations, the question that arose was how to utilize the advantage of both these data structures. One solution could have been to use both of these data structure and call them accordingly to the query. For example, the Parent relationship could have included a table for Boolean queries, such as "is 4 parent of 6" and a Parent 2D vector for getting queries, such as "get all parent of 4". The numbers listed here are statement numbers and variable indexes. However this would have resulted in unnecessary data structures saved for each relationship.

For example, Parent relationship would have:

- 2 tables to check `isParent()` and `isParentStar()`
- 2 2D vectors to store, parent and child relationships

Final Implementation

As can be seen this would result, in each relationship having 6 tables and 2D vectors which wastes a lot of space while increasing the amount of code written and decreases the ease of maintainability of the system. Therefore we have come up with something in the middle that balances the advantages and drawbacks from both of the trivial data structures shown. We have built our own generic data structure that can fit all relationships in this project. These data structures are a `ListTable` and a `MapTable`.

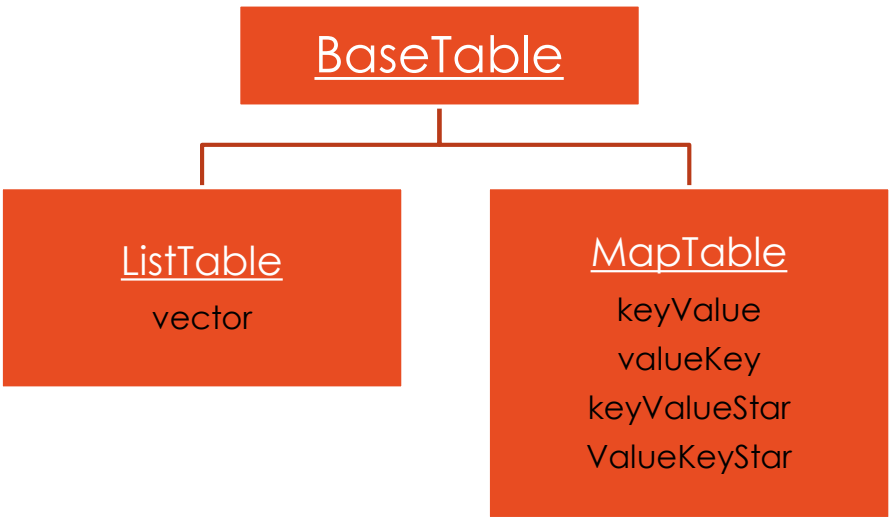


Figure 4.3: Generic Data Structure for PKB Tables

Referring to Fig 4.2, we see that the `ListTable` and `MapTable` are constructed by inheriting the base class called `BaseTable`.

The `ListTable`, has one attribute, a vector. This is used to implement tables that don't correspond to any relationship. For instance it has been implemented to build the Procedure Table, Variable Table, Constant Table and Statement Table.

The `MapTable` is more complicated, in which we have used the C++ Map data structure to implement. `MapTable` has been implemented to be of a generic type, but only accepts key and value pairs of the same data type. This has been done because of the requirement of the "star" relationship. The `MapTable` is used for the implementation of the design abstractions such as Follows/FollowsStar, Modifies, Uses, Parent/ParentStar, Call/CallStar, Next/NextStar, Contains/ContainsStar and Siblings.

Benefits of Final Implementation

1. Time Complexity Improvement:

	Is x parent of y	**Is x parent* of y**	**Get all child of x**	**Get child* of x**
Single original table	O(1)	O(1)	O(N)	O(N)
2D Vector	O(N)	O(N)	O(1)	O(1)
MapTable	O(log(N))	O(log(N))	O(log(N))	O(log(N))

Table 4.1: Comparision of Time Complexities

The time complexity table above shows a comparison of the complexities for each of the possible data structure. We see that our generic data structure much better overall for various types of queries. We note that the complexity of the `MapTable` could be further reduced if we save more information before. However for the purposes of this project, we did not want to write unnecessarily long code. As such we chose to implement this `MapTable` with this reasonable time complexity and minimized writing unnecessary code.

2. Code Maintenance & Extension:
   In the implementation of all these data tables in a single place, a developer can easily change and extend the functionality without having to duplicate a lot of code. This

reduces the chances of making mistakes. More importantly, this eases the testing since we don't have to write unit test cases for each and every single table.

As an example, here is what the Next Table could look like with the `MapTable` implementation:

```
MapTable <int> NextTable;
```

`keyValue (Map of type <int, vector<int>>)`

Possible value pairs: (1,2), (2,3), (3, 4), (4, 5)

`valueKey (Map of type <int, vector<int>>)`

Possible value pairs: (2,1), (3,2), (4, 3), (5, 4)

`keyValueStar (Map of type <int, vector<int>>)`

Possible value pairs: (1, [2, 3, 4, 5])

4.2. SPA Relationships

4.2.1. AST/Parent/Follows/ Calls

AST has a tree data structure which is implemented by us for customizations and accessibility ease. We followed strictly to the handbook's explanation of what AST should be represented as. Likewise, for parent, follows, call and call star we used the PKB to store the information while parsing the SIMPLE source. For AST, parent and follows, the implementation has not changed except for the data structure of the parent and follows table.

Modifies / Uses for procedure calls

After the modifies and uses relationship is stored in the PKB for statement numbers and variables and procedures and variables, the design extractor further extracts from these tables to cater for procedure calls with modifies and uses relationships.

Firstly, it gets the called procedures in every procedure, it then gets the modified and used variables for each of these called procedures. Then insert the used and modified variables into the called as well calling procedures and their statements. Lastly, insert the used and modified variables into the statements which have the ParentStar relationships with the calling statements.

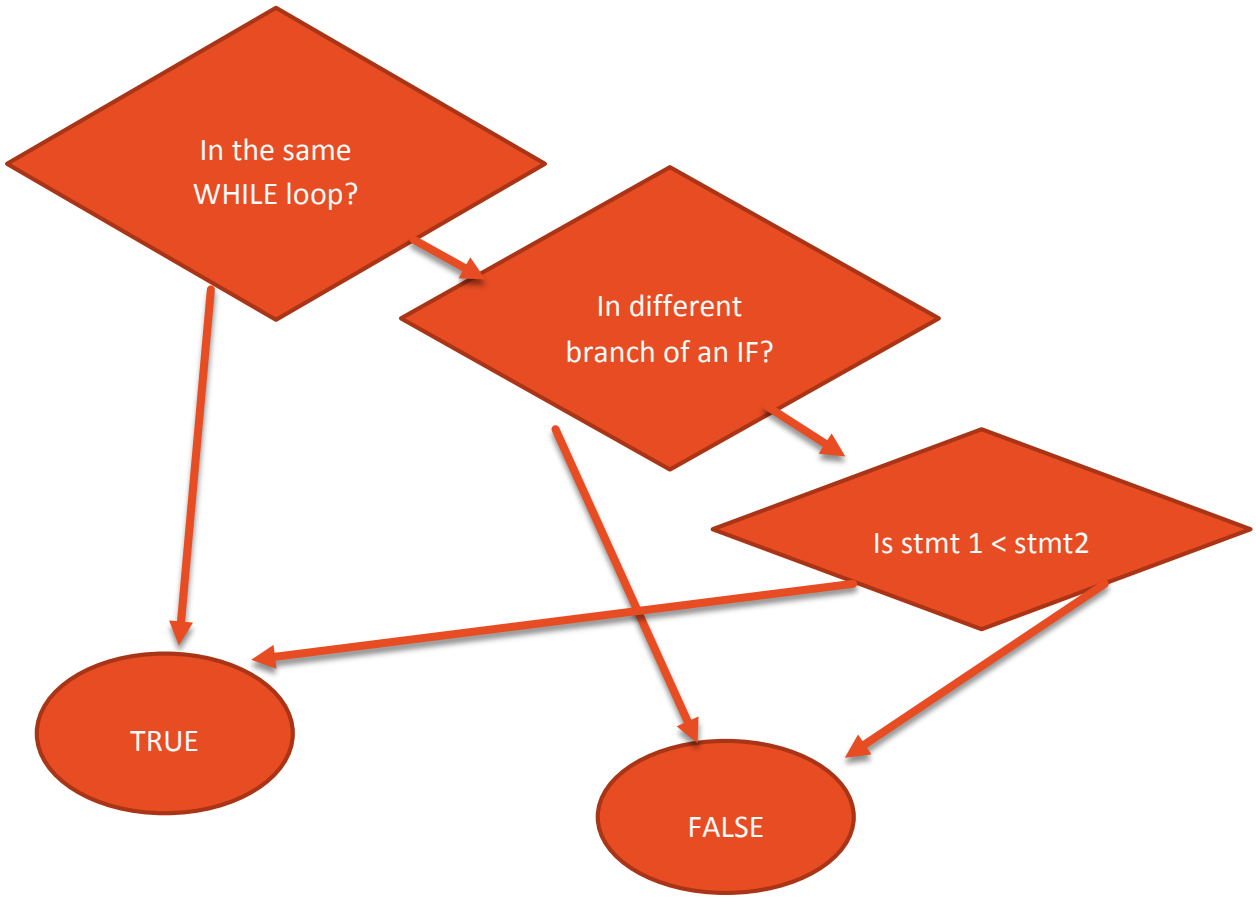
Here is the logic flow:

1. For all procedures in the SIMPLE program, get the called procedures (callStar) in this procedure. → calledProcedures
2. For every procedure in the calledProcedures vector, get the modified and used variables in it. → modifiedVar, usedVar
3. For every procedure in the calledProcedures vector, get the statements calling this procedure. → callStmts
4. Insert each modifiedVar and usedVar into the procedure to variable table for the modifies and uses relationship respectively.
5. Also insert the modifiedVar and usedVar with the callStmts into the statement to variable table for modifies and uses relationship respectively.
6. For each statement in callStmt get the parent star stmt for it and insert that statement (which has the parentStar relation to callStmt) to the statement to variable table for modifies and uses relationship respectively.

4.2.2. Next/Next Star

The Next relationship has been implemented via the `MapTable`, just like how the other relationships have been stored bases on the definition provided by the handbook. The tricky part here was in implementing the Next Star relationship. Since we were not allowed to pre-compute this relationship, we had to think of an alternative method of implementation, while keeping the complexity as low as possible. If we were to implement a trivial CFG traversing method, the time complexity for each NextStar query would have been O(N).

We have come up with an implementation that that reduces the time complexity to O(1). This design decision is another highlight of our system. The basic logic entailing our implementation is shown in the diagram below.



Re**Figure 4.4: Overview of Logic for Next* Implementation**

- If 2 statements belong to the same WHILE loop then they will lead to each other anyway.
- If not, see if they belong to the same if, and whether they on the same or different branch of that if. If they are on two different branch then we can be sure that they will never meet.
- Finally we check if statement 2 appears after statement 1 or not, if it does, then they have a Next Star relationship.

The information of the common while and if parent is saved beforehand, so the complexity in determining that is just O(1).

4.2.3. Affects/Affects Star

Affect and Affect* is purely base on CFG travel. On the path, if any assignment uses the original modified variable, that statement will be considered “Affected” by the origin of the path. The

Algorithm will stop travelling when the original modified variable is being modified by a statement on the path. By doing this, we warranty the conditions that lead to an Affect relation:

- 2 statements can lead to each other in the CFG
- Variable that be modified in statement 1 is used in statement 2
- On the path, that variable isn't modified by any statement.

An example is the figure shown below, to get statements that being affected by 1, we will travel along the CFG, 2 use x then we add 2 in the list of result, 3 modifies x then we stop travel at 3 and not go to 4.

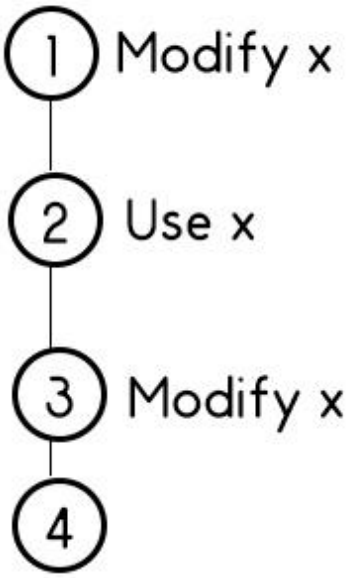


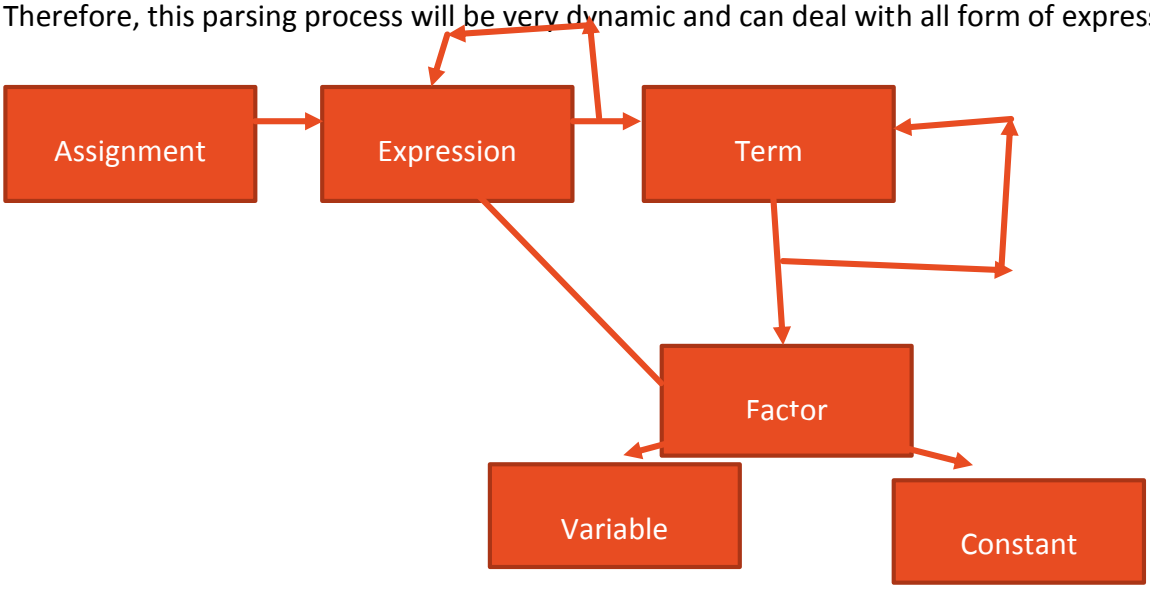
Figure 4.5: Affects Graph

Affect* is similar to affect except that we will maintain a list of modified variables, when we meet a statement that modifies a variable in this list, instead of stop traveling, we remove that variable from the list. In addition, when meet a statement that uses one of the modified variable, we add the modified variable of that statement to the list of modified variable. Beside that we also apply some minor strategies to dealing with WHILE loop because it can contain some tricky cases sometimes.

4.3. Parser

Parser is in charge of reading source files, getting tokens from source code and processing that token in an order. Parser will return an error and stop the program when it meets a syntax error in the source code. Parser will only read and store the trivial relation and the more complex ones will be handled by the DesignExtractor. Parser will not directly save data when it parses the source code but save this as temporary data first. By doing this, we are able to do unit testing for parser to see if the information it parses is correct or not before it is actually being save in the PKB.

One highlight of Parser is the reading of Assignment Statements. Our parser uses recursive parsing based on the definition of Assignment, Expression, Term, Factor in the course book. Therefore, this parsing process will be very dynamic and can deal with all form of expression.



5. Coding Standards & Experiences

In terms of the coding standards, our group has decided to adopt the following naming conventions described in this section.

5.1. Naming Conventions

5.1.1. General Rules

- Do not use underscore, hyphen or any other non-alphabet characters.
- Any name should has all the first letters of internal words capitalized, e.g. `getProcName()`
- Avoid using abbreviations. Some words are acceptable in short forms, including: *Var*, *Proc*, *Stmt*, *AST*. Other words such as *Children*, *Number* should be fully spelled out.

5.1.2. Specific Rules

- API Name:
  - API names should be nouns, in mixed case with the first letter of each internal word capitalized.
- Method:
  - Method names should be in the form of a verb. With method names containing more than one word, use mixed case with the first letter of each internal word capitalized.
  - Name of some specific methods:
    i. Methods to insert new records to the database should have the form `insertXXX()`.
    ii. Methods with return value type BOOLEAN should have the form: `isXXX()` e.g. `isExist()`, `isMatchVar()`.
    iii. Methods with return types of other values should have the form `getXXX()` e.g. `getVarName()`
    iv. Methods that return the number of records inside a table/ list should have the form `getSize()`.
    v. Methods that change the values or status of an object should have the form: `setXXX()`
    vi. Methods that return values from star queries, such as `Calls*` and `Next*`, should have the form `getXXXStar()`.

    In all of the above examples, the "XXX" is used in place of the specific name which the method will adopt.

To keep the abstract and concrete PKB API in sync, we created a variable table, statement table and procedure tables. These tables are vectors mapping variable names to indexes of the vectors. So that a API method like `BOOLEAN isModifies(STMT_NUM s1, INDEX varIndex)` understands that `INDEX` is the mapped value of a certain variable name, where `INDEX` is just an integer in C++ type.

6. Query Processing

The query processing, the second major component of the SPA, handles the management and evaluation of various queries based on the Simple source parsed by the Parser. As is seen in the system architecture diagram in Fig [num], the QP contains components that we have added on top of the suggested components from the handbook. This would be the Query Representator (QR) and the Query Optimizer (QO). The reasoning for their addition and their functionality shal be discussed under the section for Query Evaluation (Section 6.2).

In this section we will go through how queries are first of all validated and then evaluated. For the discussion we will be referring to the sample query, shown below, to illustrate the process.

> assign a; call c; while w; variable v;
>
> Select <c.procName, a> such that Parent*(w, a) and Uses(a, v) pattern a(_, _"2*y + 3"_) with c.procName = v.varName

6.1. Query Validation

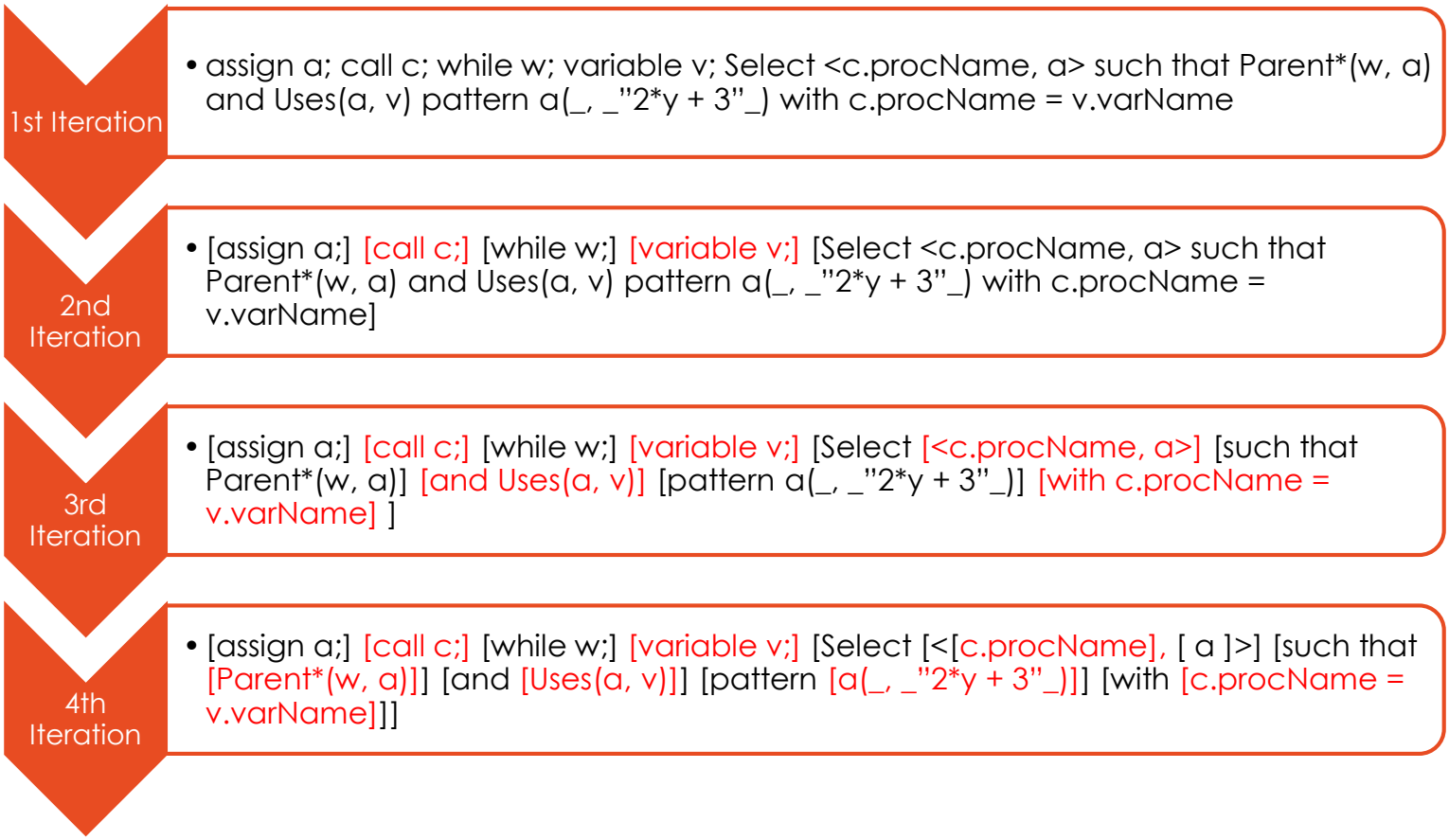
The query validation process is handled by the QueryPreprocessor (QPP). The QPP receives each query in a string form, and tries to read it from left to right, until the end of the string. During reading, if it meets a block of symbols which fits a PQL grammar rule, the QPP breaks the block apart from the string and validates that block using the corresponding grammar rule. This reading and breaking validating process continues until no more new block of symbols can be found. This is an example of how we have used a top-down approach to tackle this component.

For each undivided block of symbols, QPP will validate it using the grammar rules shown below. If any rule is violated, the QPP will send back information to the QR to handle this error. Otherwise, the information of the block will be saved into the respective components of the QR.

Grammar rules to be checked, line by line:

- On finding a line containing a declaration (e.g. stmt s;), QPP will send this part to the `preprocessDeclaration()` method.
- On finding a line containing a query part (e.g. Select s such that Follows(s, 1)), QPP will send this part to `preprocessQueryPart()` method.
- While the `preprocessQueryPart()` method runs, if the QPP finds a new clause of query, it will call the corresponding method `preprocessClause()` (e.g. `preprocessSuchThatCondtion()`) for this clause. Currently, we have provided the methods for "*such that*" clause and "*pattern*" clause.

Based on the sample query shown above, the validation process is shown below:



In each iteration as a new block of symbols are found, they are enclosed by a square bracket. From iteration 2 onwards, the specific blocks have been colored black and red alternatively to show the distinction of the discovered blocks. This process continues on until everything has been validated. For each block of symbols, QPP will call the function corresponding to the grammar rule for this block, such as `preprocessDeclaration()` for [assign a;] or `preprocessAttrRef()` for [c.procName].

6.2. Query Evaluation

6.2.1. Data Representation (QR)

As was mentioned above, we have implemented a new component called the QR. This component basically stores all of the necessary information of the query, from the validation process, for the evaluation process. We decided to include a QR to deal with the practice of

information hiding. With the abstraction of a new component, handling all of this data storage, it would be easier to separate the various concerns at each part of the query processing. Since a query contains two parts: a list of symbol declarations and the main query itself, the QR saves that information into 2 components of the QR:

1. SymbolTable: Used for the storing of all the declarations in the query. A declaration is separated and saved into 2 parts: the declared entity and symbol name.
2. QueryTree: Used for storing of the query itself. Starting from the keyword: "Select" until the end of the query. Each symbols of the query part is stored into a node, `TNode` and linked with others to form a query tree.
3. In addition to these 2 data structures, the QR also saves a `BOOLEAN` value to indicate whether the query is free of grammar errors or not. Later the QE will check this value first to decide if it should ignore the query evaluation in case the query is grammatically incorrect.

Shown below is how the sample query would look like in the QR.

SymbolTable:

Id	Type	Name
0	assign	a
1	call	c
2	while	w
3	variable	v

Figure 6.1: SymbolTable for Sample Query

QueryTree:

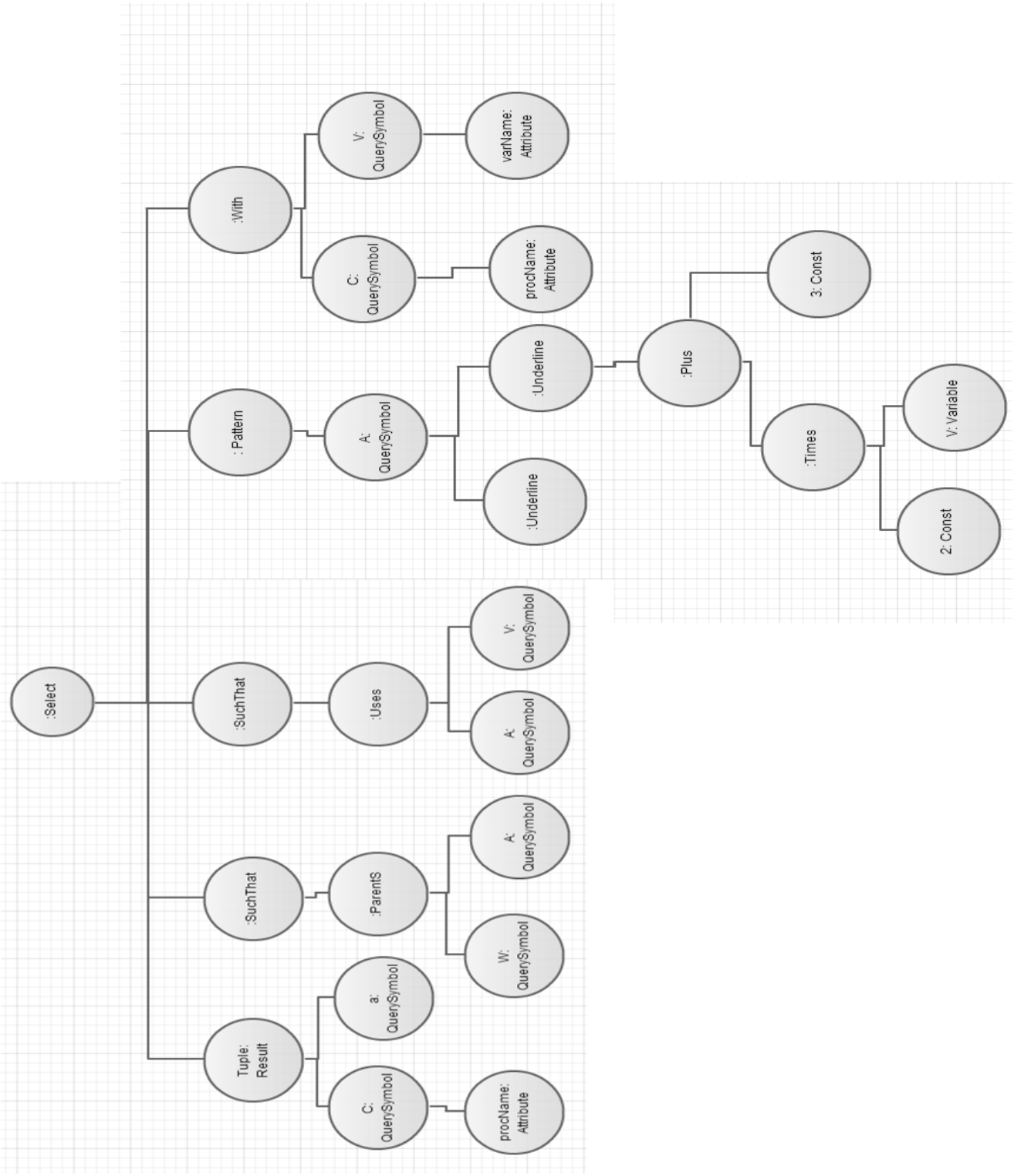


Figure 6.2: Query Tree of sample query

Once all of this information has been stored inside the QR, we can move on with the actual query evaluation, which is handled by the QE.

6.2.2. Basic Query Evaluation

6.2.2.1. *Manage the temporary results*

It was noted in the old version of the SPA, in CS3201, that we did not have to worry about dealing with managing temporary results, since the queries did not contain multiple clauses. As a result, the main concern that we had was on how to extend the QE to deal with the management of temporary results.

Initial Implementation

The first method that we had implemented to manage temporary results was through a temporary vector inside of the QE. During the evaluation process, this vector would continuously be updated, for each clause until there is no more unsolved clause or an unsatisfiable clause is met. The QE would then use this vector to update the final result, and destroy this temporary result vector after that.

This naïve solution was sufficient to get us started on evaluating simple multiple clauses. However as more clauses were considered in a query, and as each query would get more complex, it was obvious that this solution was clearly insufficient. The main limitation of this solution was that it does not allow for a review of the temporary results, as they are discarded. This is difficult for programmers during debugging and updating the program. Moreover, as the QE only uses one vector to store temporary data and solving the various all clauses, it raises the complexity of the evaluation process. Also not every data from each temporary result is needed in solving a certain clause.

Using the sample query above, we figured out that the time complexity, using this implementation, would be $O(N^4 P)$ where N is the number of values for a query symbol and P is the time for a PKB's operation. It is obvious that this complexity is highly costly and not efficient.

Actual Implementation

Given the aforementioned issues, we came up with a new solution that included the use of using tables to store and manage temporary results: a dynamic `ResultTable` and `ResultManager`. The `ResultTable` keeps the list of values for symbols used in the query and the `ResultManager` maintains the list of `ResultTable`. The `ResultManager` interacts with the QE during evaluation to record and return data when the QE demands. The main features of the `ResultManager` are:

- Returning of concise and non-duplicated data to the QE. Before evaluating a clause, the QE will ask the `ResultManager` for data relating to that clause. The `ResultManager`, after receiving a list of symbols used in the clause, will extract the data required from all `ResultTable` in the list. This extracted data is of the given symbols only, for a particular clause, and no data is duplicated.

- Minimize the space complexity as much as possible: After evaluation, a new `ResultTable` will be sent from QE to `ResultManager`. Before saving it to the `ResultTable` list, `ResultManager` will try to merge the existing tables with the new one by checking for shared symbols. If there are symbols shared between an existed table and the new table, `ResultManager` will merge those two tables together. Otherwise, `ResultManager` will do nothing. In this way, each symbol will only be saved once in the table, and non-related symbols (symbols which are not used in the same clause) will be kept in different tables. This approach cuts down the time and space needed to maintain and extract data from `ResultManager`.

With this new approach in mind, the symbols' data are extracted from the `ResultManager` before evaluation. This means that the cost of solving each clause is $O(N^2P)$ at most. After that, when we insert the new `ResultTable` back to the `ResultManager`, the time complexity is $O(N^4M)$ for merging, where M is the merging time for 2 rows. However, since the number of values in a row is usually small, we can safely assume that $M<<P$. Thus, in comparison with the old approach, this new approach is definitely faster and more efficient.

Based on the sample query, the temporary result management is illustrated below. The illustration starts from evaluating the pattern clause in the query, assuming that the previous clauses have been evaluated, with some temporary results stored in the `ResultManager`

1. Evaluation of query is now at "pattern a(_, _"2*y + 3"_)". At that time, `ResultManager` contains 1 `ResultTable`.

Id	w	a	v
0	1	2	x
1	1	4	t
2	1	8	y
3	3	4	t
4	3	8	y
5	5	8	y

Table 6.2: ResutTable in ResultManager

2. To solve the `pattern` clause, QE asks `ResultManager` to extract the values of symbol "a". `ResultManager` will return the following table:

Id	a
0	2
1	4
2	8

Table 6.1: Table returned to QE

3. After evaluation, the table now contains only data satisfying this pattern clause.

Id	a
0	8

Table 6.3: Updated Table for values of "a"

(Continued on the next page)

4. The QE then asks the `ResultManager` to insert this new table into its table list. `ResultManager` must merge the existing table, in 1, with this new one, in 3. The table below is the final merged table.

Id	**w**	**a**	**v**
0	1	8	y
1	3	8	y
2	5	8	y

Table 6.4: Final Result after evaluating Pattern

6.2.3. Optimization

The optimization process for the evaluation takes place after the validation in the QPP and before the evaluation in the QE. Since this part of the optimization is separated from the other functionalities, we decided to implement a Query Optimizer (QO). After all of the information has been stored inside of the QR, the QO is called to manipulate the results in the `SymbolTable` and `QueryTree` for optimization. The QO is in charge of the following 2 duties:

1. Rank all clauses of the given query based on 2 features:
    a. The clause's type
    b. The number of query symbols used in each clause.

   In our program, we give a higher rank to "`with`" clause, followed by the "`such that`" and "`pattern`" clauses. For the second feature, the higher the number of query symbols, the lower a rank the clause will have. For instance, a clause using 2 query symbols will have a lower ranking than another with 2 underlines and 1 query symbol.

2. Sort the query tree again using the ranking of the clauses. We assume that the clauses with a higher ranking will have a better time cost during evaluation, and thus we try to solve those clauses first. After the sorting, the query tree will have its clause nodes re-arranged, with the higher-ranked clause node in the front, following by the lower ones.

Referring to Fig 6.2 of the query tree, the order of the nodes, after optimization would be as follows:

Such-that - > such-that -> pattern-> with [Before Optimization]

Pattern -> with -> such-that -> such-that [After Optimization]

The pattern clause, while having a low ranking for its type, is placed first since it only contains 1 query symbol. For the rest of the clauses, each of them uses 2 query symbols, thus the order is mainly based on their types.

6.2.4. Design Decisions

This section outlines how we have implemented each of the components of the QP. It mainly shows what data structure we have used.

Query Representator

The `SymbolTree` and `QueryTree` have been implemented by inheriting from the parent class `Tree`, which is also used for building the AST. By using the same data structure, we can compare 2 trees easily. This is useful for when we have to solve the pattern clause.

Query Optimizer

No specific data structure has been used. The data required is taken from the QPP after validation. There are 2 main functions in QO:

- `RankTree()` : Gives a ranking score to any clause node met during travelling the tree. The method applies a depth first search.
- `SortTree()` : Based on the rankings, QO will sort the clauses node from a high ranking to lower ones. This method uses merge sort for sorting.

Temporary Result Storage

- `ResultTable`

  Saves a list of temporary results in a table form. The `ResultTable` provides basic methods to insert/ get/ delete data of the table, and one more method call `extractTable(vector<string> symbols)` to extract data of symbols in the input.

- `ResultManager`

Saves a list of pointers to different `ResultTable`. It provides 2 methods to insert and extract data from its tables.

7. Testing

7.1. Testing Experience

The testing of the SPA was carried out in three stages. Firstly, for each component of the PKB and PQL, we carried out unit testing. Unit testing allowed us to test the internal functions of each component. After unit testing was completed, we carried out integration testing of the different components. This ensured that the different components work properly together, for instance, the Parser and the Query Evaluator. Once we tested specific components, we then tested the system as a whole in validation testing. Keeping in mind that his project was implemented in iterations, we carried out this 3 stage testing procedure each time a new functionality was implemented. Even if it was for a new function in a preexisting component.

The most vigorous type of testing that was carried out was validation testing. We tested the system with hundreds of test cases and many complex source codes. The queries that we used to test covered a range of possibilities. From the most basic, to boundary cases, to where the thing to be returned could not be found in any of the tables.

7.2. Examples of Test Cases

7.2.1. Unit Testing

The aim of unit testing, would be to discover any logical errors present in the code. This allowed us to pinpoint specific errors, saving us the hassle of running into such errors during further stages of testing. Unit testing was done by manually inserting values and asserting that the function outputs were as expected.

The following example illustrates the unit testing of the `ListTable` component.

```
void ListTableTest::TestGetIndexes() {
    ListTable <string> listTable;
    string element1 = "a";
    string element2 = "b";
    string element3 = element1;
    listTable.insert (element1);
    listTable.insert (element2);
    listTable.insert (element3);

    int expectedResultSize = 2;
    int actualResultSize = listTable.getIndexes(element1).size();

    CPPUNIT_ASSERT_EQUAL (expectedResultSize, actualResultSize);
}
```

Inserts duplicate elements into the table

Ensures that table does not record duplicate element

Figure 7.1: Unit Test Example

7.2.2. Integration Testing

Many of the components work well when tested individually. However when they are integrated with other components, unforeseen errors may appear. This is the aim of integration testing, to identify such errors between the interactions of components. Such errors, in our case, were attributed to the fact thtt the PKB, Parser and PQL were written by different members. Hence the components had slightly different methods of implementation and different expected inputs and outputs. During integration testing, these flaws between the components became apparent to us. The Parser parses the simple source code, which is provided as the input file, and the PKB

is built from it. In the following example, the integration between these components, including the Design Extractor, is illustrated when testing the Calls relationship.

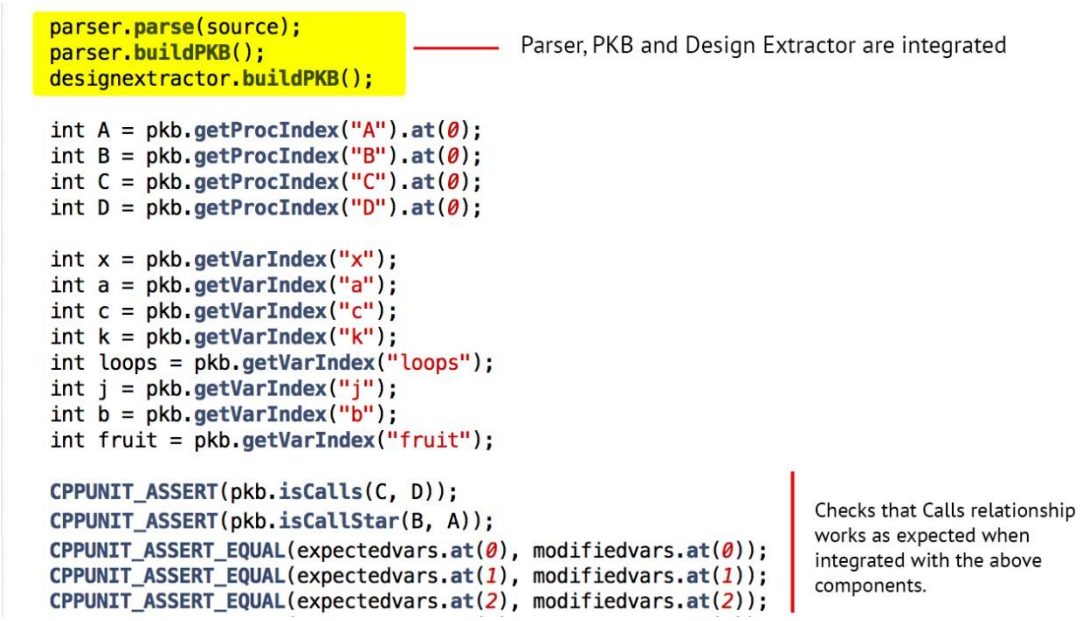


Figure 7.2: Integration Test Example

7.2.3. Validation Testing

Validation testing, also known as System testing, is crucial to test the SPA system as a whole. There are many logical loopholes that might have been missed during unit and system testing, which can be spotted during extensive system testing. Thus, it is important to have a variety of test cases and source codes. The aim, in our case, was to ensure that the system can handle queries of multiple complexities and parse hundreds of lines of source code, including those haphazardly formatted.

The following Simple source code aims to test complex pattern queries by using multiple expressions and variables, the heavy use of brackets is also to be noted.

```
procedure Pattern {
a = i+(n-2);
i= a+(5-3)*n;
b=(a+5)+1-(a-n)*c-(i+o);
n=i-((n+a)*o)+c;
b=(((a+5)+1)-(a-n))*c-(i+o);
}
```

Figure 7.3: Simple Source for Validation Testing

The figure below shows two query examples. The first query tests the combination of multiple relationship clauses. This tests that the evaluated results for the different clauses are merged correctly to output only the correct answers that satisfy the whole query. This is essential for evaluation of any query that has more than one clause. The second query focuses on selecting variables by their attributes, such as 'stmt#' and 'varName'. This is a commonly used selection method in queries, hence it's essential to ensure that all such possible attribute selections have been addressed by the system.

```
Multiple_Clauses, While, Uses, Parent ::
while w1,w2,w3,w4,w5;
1 Select w1 such that Parent(w1,w2) such that Parent(w2,w3) such that Parent(w3,w4) such
that Modifies(w4,"z") such that Uses(w4,"z")
82, 88

Affects*,assignment :: s1
assign a1,a2; variable v1,v2,v3,v4,v5;
2 Select <a1,v1> such that Affects*(a1,a2) with a2.stmt#=54 and v1.varName=v2.varName
and v2.varName=v3.varName and v3.varName=v4.varName and v4.varName=v5.varName with v5.
varName="q"
49 q
```

Figure 7.4: Queries for Validation testing

8. Discussion

8.1. Possible Improvements to Project

Due to time constraints, we were unable to implement the bonus features for the SPA. Instead, we focused on making our basic requirements more robust. However we did manage to implement the Contains and siblings relationship for the basic cases.

If we had a chance to start all over again, we would start with making the code open to extensions but closed to modifications. We would also apply several other design principles right from the start. Presently, we had to change quite a bit of our code due to inefficient planning. Even though for the better, that cost us quite a lot of time.

Currently, we have implemented the Contains and Siblings relationship in the design extractor where the AST is traversed and these relationships are extracted. However, due to time constraints, we were not able to integrate these clauses in the Query Evaluator. We wanted to ensure that the basic requirements were working well before concentrating of the bonus features.

8.2. Tools used

The main tools we used for the project were Visual Studio 2010, CPPUnit library and the Autotester. CPPUnit was very useful as it proved as an essential tool during the testing of the system. In addition, we used Github as our revision control system. This was essential in helping to keep track of the work done by the different members and merging them together in the end.

8.3. Lessons Learnt

Each iteration of the project has a deadline of a few weeks, hence we learned how to manage our time such that we meet the requirements for each iteration. However we also learnt that it is alright to delay moving on to the next iteration if we did not have a fully working version of the current iteration yet. The quality of the work is more important than finishing up it up in a rushed manner in order to meet the deadlines. This would just result in unresolved errors and ill-structured code with poor logic. These flaws would carry on to the next iteration and would end up as bigger problems. Hence, we learnt that it is better to have well-tested basic features rather than many features that have many errors.

We also took on our tutor's advice to criticize each other's work before implementing the feature. This helped us find a lot loop holes in our teammate's solutions and saved us a lot of debugging time. We also learnt from each other in the process.

Initially we had divided components such as the PKB, Query Evaluator and Parser to one person each. However, soon we realized that it is more efficient to break the components down further and allow different team members to implement them so that everyone is aware of the logic in each component. An example is to break the Query Evaluation part further to Query Pre - processor and Result Manager. This also allows mistakes to be found earlier. As another team member is more likely to find faults in another team member's work.

Appendix A: Abstract PKB API

PKB
Overview: The PKB contains all the components required for the storage of data. Such as the tables and AST.
Public Interface:
BOOLEAN *isFollows* (**STMT_NUM** s1, **STMT_NUM** s2) **Description**: Method to return if statement s1 is followed by statement s2. Return true if relationship holds, otherwise return false.
BOOLEAN *isFollowsStar* (**STMT_NUM** s1, **STMT_NUM** s2) **Description:** Method to check Follows*(s1, s2) holds. Return true if relationship holds, otherwise return false.

BOOLEAN *insertFollows* (**STMT_NUM** s1, **STMT_NUM** s2)

Description:

Method to insert a pair of following statement numbers in FollowTable. Return true if successful, otherwise return false.

STMT_NUM *getFollowingStmt* (**STMT_NUM** s1)

Description:

Method to get the following statement to statement number s1.
Return the statement number if found, otherwise return -1.

STMT_NUM *getFollowedStmt* (**STMT_NUM** s1)

Description:

Method to get statement which is followed by statement s1.
Return the statement number if found, otherwise return -1.

VECTOR<STMT_NUM> *getFollowingStarStmt* (**STMT_NUM** s1)

Description:

Method to get the list of following statements to statement number s1 with relationship Follows*.

Otherwise, return NULL.

VECTOR <STMT_NUM> *getFollowedStarStmt* (**STMT_NUM** s1)

Description:

Method to get the list of statements which are followed star by statement s1.

Otherwise, return NULL.

BOOLEAN *isModifies* (**STMT_NUM** s1, **INDEX** varIndex)

Description:

Method to check if modifies relationship exists. Return true if exists, otherwise return false.

BOOLEAN *insertModifies* (**STMT_NUM** s1, **INDEX** varIndex)

Description:

Method to insert a pair of statement number and variable holding the Modify relationship in ModifyTable.
Return true if successful, otherwise return false.

VECTOR<INDEX> *getModifiedVarAtStmt* (**STMT_NUM** s1)

Description:

Method to get the variables modified in statement s1.
Otherwise, return NULL.

VECTOR<STMT_NUM> *getStmtModifyingVar* (**INDEX** varIndex)

Description:

Method to get the list of statements that modify var.
Otherwise, return NULL.

BOOLEAN *isModifiesProc* (**PROC** proc1, **INDEX** varIndex)

Description:

Method to check if modifies relationship exists. Return true if exists, otherwise return false.

BOOLEAN *insertModifiesProc* (**PROC** proc1, **INDEX** varIndex)

Description:

Method to insert a pair of procedure and variable holding the Modify relationship in ModifyTable.
Return true if successful, otherwise return false.

VECTOR<INDEX> *getModifiedVarAtProc* (**PROC** proc1)

Description:

Method to get the variables modified in procedure proc1.
Otherwise, return NULL.

VECTOR<PROC> *getProcModifyingVar* (**INDEX** varIndex)

Description:

Method to get the list of procedures that modify var.
Otherwise, return NULL.

INDEX *insertVar* (**STRING** VarName)

Description:

If “varName” is not in the VarTable, insert it into the VarTable and return its index value.
Otherwise, return -1 (special value) and the table remains unchanged.

STRING *getVarName* (**INT** index)

Description:

If there is record in VarTable having index value “index”, return its variable name.

If “index” is out of range:

Throws: InvalidReferenceException

INDEX *getVarIndex* (**STRING** varName)

Description:

If there is record in VarTable having name “varName”, return its index value.

Otherwise, return -1 (special value)

INT *getVarTableSize* ()

Description:

Returns the size of the VarTable.

BOOLEAN *isParent* (**STMT_NUM** s1, **STMT_NUM** s2)

Description:

Method to return if statement s1 is parent of statement s2. Return true if relationship holds, otherwise return false.

BOOLEAN *isParentStar* (**STMT_NUM** s1, **STMT_NUM** s2)

Description:

Method to return if statement s1 is parent star of statement s2. Return true if relationship holds, otherwise return false.

BOOLEAN *insertParent* (**STMT_NUM** s1, **STMT_NUM** s2)

Description:

Method to insert a pair of parent and child statement numbers in ParentTable.
Return true if successful, otherwise return false.

VECTOR<STMT_NUM> *getChildStmt* (**STMT_NUM** parentStmt)

Description:

Return an array of child statement numbers of parent statement if found in ParentTable.
Otherwise return NULL.

VECTOR<STMT_NUM> *getParentStarStmt* (**STMT_NUM** childStmt)

Description:

Return an array of statement numbers that are parent star of child statement if found in ParentTable.
Otherwise return NULL.

VECTOR<STMT_NUM> *getChildStarStmt* (**STMT_NUM** parentStmt)

Description:

Return an array of child statement numbers of parent star statement if found in ParentTable.
Otherwise return NULL.

INT *getParentTableSize()*

Description:

Returns the number of records in ParentTable.

BOOLEAN *insertStmt* (**STRING** name)

Description:

Inserts statement in StatTable.
Return true if successful, otherwise return false.

VECTOR<STMT_NUM> *getStmtIndex* (**STRING** name)

Description:

Return index of statement having name in StatTable.
Otherwise, return NULL.

STRING *getStmtName* (**INDEX** index)

Description:

Return name of statement having index in StatTable.
Otherwise, return NULL.

INT *getStatTableSize()*

Description:

Returns the number of records in StatTable.

BOOLEAN *isUses* (**STMT_NUM** s1, **INDEX** varIndex)

Description:

Method to check if uses relationship exists. Return true if exists, otherwise return false.

BOOLEAN *insertUses* (**STMT_NUM** s1, **INDEX** varIndex)

Description:

Method to insert a pair of statement number and variable holding the Use relationship in UseTable.
Return true if successful, otherwise return false.

VECTOR<INDEX> *getUsedVarAtStmt* (**STMT_NUM** s1)

Description:

Method to get the variables used in statement s1.
Otherwise, return NULL.

VECTOR<STMT_NUM> *getStmtUsingVar* (**INDEX** varIndex)

Description:

Method to get the list of statements that use var.
Otherwise, return NULL.

BOOLEAN *isUsesProc* (**PROC** proc1, **INDEX** varIndex)

Description:

Method to check if uses relationship exists. Return true if exists, otherwise return false.

BOOLEAN *insertUsesProc* (**PROC** proc1, **INDEX** varIndex)

Description:

Method to insert a pair of procedure and variable holding the Use relationship in UseTable.
Return true if successful, otherwise return false.

VECTOR<INDEX> *getUsedVarAtProc* (**PROC** proc1)

Description:

Method to get the variables used in procedure proc1.
Otherwise, return NULL.

BOOLEAN *insertConst* (**STRING** name)

Description:

Inserts constant in ConstTable.
Return true if successful, otherwise return false.

BOOLEAN *isConst* (**STRING** name)

Description:

Method to check is name is a Constant.
Return true if successful, otherwise return false.

INDEX *getConstIndex* (**STRING** name)

Description:

Return index of constant having name in ConstTable.
Otherwise, return NULL.

STRING *getConstName* (**INDEX** index)

Description:

Return name of constant having index in ConstTable.
Otherwise, return NULL.

INT *getConstTableSize()*

Description:

Returns the number of records in ConstTable.

BOOLEAN *insertProc* (**STRING** name)

Description:

Inserts procedure in ProcTable.
Return true if successful, otherwise return false.

BOOLEAN *isProc* (**STRING** name)

Description:

Method to check is name is a procedure.
Return true if successful, otherwise return false.

INDEX *getProcIndex* (**STRING** name)

Description:

Return index of procedure having name in ProcTable.
Otherwise, return NULL.

STRING *getProcName* (**INDEX** index)

Description:

Return name of procedure having index in ProcTable.
Otherwise, return NULL.

INT *getProcTableSize()*

Description:

Returns the number of records in ProcTable.

BOOLEAN *isCalls* (**PROC** proc1, **PROC** proc2)

Description:

Method to check if calls relationship exists. Return true if exists, otherwise return false.

BOOLEAN *insertCalls* (**PROC** proc1, **PROC** proc2)

Description:

Method to insert a pair of procedure numbers holding the Call relationship in CallTable. Return true if successful, otherwise return false.
VECTOR<PROC> *getCalledByProc* (**PROC** proc1) **Description:** Method to get the procedure called in procedure proc1. Otherwise, return NULL.
VECTOR<PROC> *getCalledByStarProc* (**PROC** proc1) **Description:** Method to get the list of procedures that call star proc1. Otherwise, return NULL.
BOOLEAN *isCallStmt* (**PROC** proc1, **STMT_NUM** s1) **Description:** Method to check if calls relationship exists. Return true if exists, otherwise return false.
BOOLEAN *insertCallStmt* (**PROC** proc1, **STMT_NUM** s1) **Description:** Method to insert a pair of procedure and statement holding the Call relationship in CallTable. Return true if successful, otherwise return false.

VECTOR<PROC> *getCallingProc* (**PROC** proc1)

Description:

Method to get the procedures called in procedure proc1.
Otherwise, return NULL.

VECTOR<PROC> *getCallingStarProc* (**PROC** proc1)

Description:

Method to get the procedures called in procedure proc1 with star relationship.
Otherwise, return NULL.

PROC *getCalledProc* (**STMT_NUM** s1)

Description:

Method to get the procedure called in statement s1.
Otherwise, return NULL.

VECTOR<STMT_NUM) *getCallingStmt* (**PROC** proc1)

Description:

Method to get the statement calling procedure.
Otherwise, return NULL.

BOOLEAN *isNext* (**LINE_NUM** n1, **LINE_NUM** n2)

Description:

Returns true if n2 is next to n1. Otherwise false.

INDEX *insertNext* (**LINE_NUM** n1, **LINE_NUM** n2)

Description:

If the relation Next(n1, n2) is not in Next Table, insert it into the table and return the sizeof the table.

Otherwise: return -1 (special value) and the table remains unchanged.

VECTOR<STMT_NUM> *getNextStmts* (**STMT_NUM** s1)

Description:

Returns vector of stmt numbers next to s1.

VECTOR<STMT_NUM> *getPreviousStmts* (**LINE_NUM** n1)

Description:

Returns vector of stmt numbers previous of n1.

BOOLEAN *isNextStar* (**LINE_NUM** n1, **LINE_NUM** n2)

Description:

If there is no record of relation Next() of line numbers n1 and n2 return FALSE.

Otherwise return TRUE.
VECTOR<STMT_NUM> *getNextStarStmts* (**LINE_NUM** n1) **Description:** Returns vector of next to n1 with the star relationship
VECTOR< STMT_NUM > *getPreviouStarStmts* (**LINE_NUM** n1) **Description:** Returns vector of previous to n1 with the star relationship
BOOLEAN *isAffect* (**STMT_NUM** s1, **STMT_NUM** s2) **Description:** Method to check if affects relationship exists. Return true if exists, otherwise return false.
BOOLEAN *isAffectStar* (**STMT_NUM** s1, **STMT_NUM** s2) **Description:** Method to check if affects star relationship exists. Return true if exists, otherwise return false.
VECTOR<STMT_NUM> *getAffected* (**VECTOR<VARIABLES>** modifiedVar, **STMT_NUM** currentStmt, **VECTOR<STMT_NUM>** path)

Description:

Method to get the statements affected by statement currentStmt. Recursive function. Otherwise, return NULL.

VECTOR<STMT_NUM> *getAffecting* (**VECTOR<VARIABLES>** modifiedVar, **STMT_NUM** currentStmt, **VECTOR<STMT_NUM>** path)

Description:

Method to get the statements affected by statement currentStmt. Recursive function. Otherwise, return NULL.

VECTOR<STMT_NUM> *getAffectedStar* (**STMT_NUM** s1)

Description:

Method to get the affected star relationship statements Otherwise, return NULL.

VECTOR<STMT_NUM> *getAffectingStar* (**STMT_NUM** s1)

Description:

Method to get the affecting star relationship statements Otherwise, return NULL.

BOOLEAN *isSibling* (**NODEID** nodeID1, **NODEID** nodeID2)

Description:

Method to check if 2 nodeID are siblings in the AST.

Return true if successful, otherwise return false.

VECTOR<NODEID> *getSibings* (**NODEID** nodeID1)

Description:

Return nodeid of siblings of nodeID1.
Otherwise, return NULL.

BOOLEAN *insertSibling* (**NODEID** nodeID1, **NODEID** nodeID2)

Description:

Return true if insertion into the Siblings table is successful.

Otherwise, return false.

INT *getSiblingTableSize()*

Description:

Returns the number of records in Siblings Table.

BOOLEAN *isContains* (**NODEID** nodeID1, **NODEID** nodeID2)

Description:

Method to check if 2 nodeID are contained in the AST.

Return true if successful, otherwise return false.

VECTOR<NODEID> *getContained* (**NODEID** nodeID1)

Description:

Return nodeid of containing by nodeID1.
Otherwise, return NULL.

VECTOR<NODEID> *getContaining* (**NODEID** nodeID1)

Description:

Return nodeid of contained by nodeID1.
Otherwise, return NULL.

BOOLEAN *insertContains* (**NODEID** nodeID1, **NODEID** nodeID2)

Description:

Return true if insertion into the Contain table is successful.

Otherwise, return false.

INT *getContainTableSize()*

Description:

Returns the number of records in Siblings Table.

VECTOR<NODEID> *getContainedStar* (**NODEID** nodeID1) **Description:** Return nodeid of containing by nodeID1. Otherwise, return NULL.
VECTOR<NODEID> *getContainingStar* (**NODEID** nodeID1) **Description:** Return nodeid of contained by nodeID1. Otherwise, return NULL.

Appendix B: Comments on Handbook

Overall, we found the handbook very helpful in giving us ideas for the implementation of the SPA. One thing that could have been better was if there had been more examples and perhaps some sample exercises for us to practice our concepts on. The extensions such as Contains and Siblings could also be explained briefly in the handbook instead of introducing them later on. This means we would have rough idea of how these relationships work beforehand and when it is time to consider whether or not to implement them, we can focus on finding the best way to integrate them rather than spending time understanding and grasping the new concepts of these relationships, which reduces crucial time which could be spent on the actual implementation.