National University of Singapore
School of Computing

CS3202: Software Engineering Project II

TEAM 05: Flying Cockroach

Semester 1, AY2014/2015

Matriculation Number	HP Number	Student Name	Email
Group-PQL:			
A0099214B	8518 2707	Adinda Ayu Savitri	savitri.adinda@gmail.com
A0098139R	9082 0864	Hisyam Nursaid Indrakesuma	indrakesuma.hisyam@gmail.com
A0103494J	9620 7018	Lacie Fan Yuxin	lacie.jolene.fan@gmail.com
Group-PKB:			
A0101286N	9833 2474	Ipsita Mohapatra	ipsita@nus.edu.sg
A0080415N	9148 6248	Steven Kester Yuwono	a0080415@nus.edu.sg
A0099768Y	9178 6540	Yohanes Lim	yohaneslim93@gmail.com

Consultation Day/Hour: Monday 1pm

CONTENTS

1. SPA

Static Program Analyser (SPA) is a program to answer queries about an input SIMPLE program. In this report, we will be describing the design and implementation decisions made during the development of the basic SPA and any bonus features.

1.1. ARCHITECTURE

The architecture consists of 3 main components: the Code Parser, the PKB and the Query Processor. Code Parser parses the code and stores design abstractions in each of the 8 tables in the PKB. After Query Parser has parsed the query, the Query Evaluator consults the PKB API to answer queries.

Both the Code Parser and the Query Processor are dependent on PKB but not dependent on each other. This ensures low coupling between components within SPA. At the same time there is high cohesion within each component, as described in the subsequent parts.

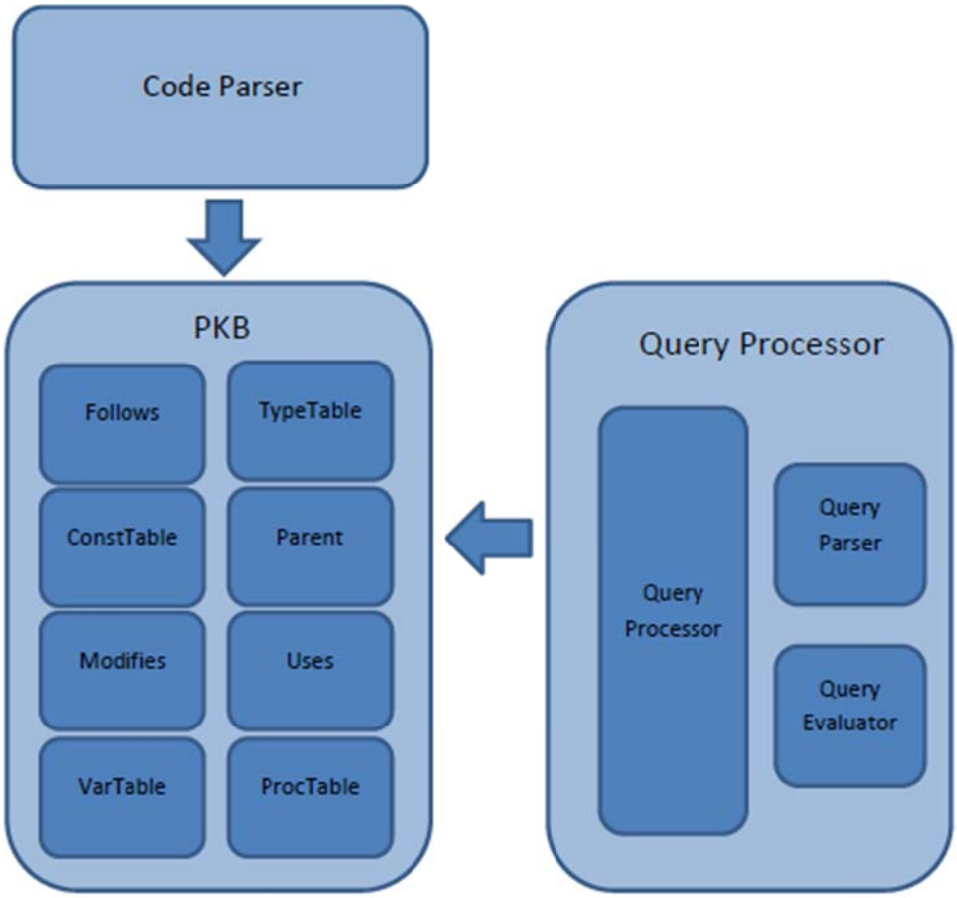

Figure 1

1.2. Development Plan

1.2.1. For Whole Project

	Iteration 1					Iteration 2			
Team member	**Implement Calls in PKB**	**Implement Modifies and Uses in PKB**	**Extend Query Parser to support with clause**	**Extend Query Evaluator to support with clause**	**Write system test cases for the enhancement**	**Implement Next in PKB**	**Extend Query Parser to support multiple clauses**	**Extend Query Evaluator to support multiple clauses**	**Write system test cases for the enhancement**
Adinda									
Lacie									
Hisyam									
Steven									
Ipsita									
Yohanes									

	Iteration 3			
Team member	**Implement Affects in PKB**	**Extend Query Parser to support Affects and tuple results**	**Extend Query Evaluator to support Affects and tuple results**	**Write system test cases for the enhancement**
Adinda				
Lacie				
Hisyam				
Steven				
Ipsita				
Yohanes				

1.2.2. For Iteration 3

	Iteration 3								
Team member	**Data hiding for PKB components and method simplification**	**Revamp PKB data structures to improve performance**	**Next Star optimization by implementing nextPair relationship**	**Unit testing for nextPair**	**Extend Query Parser to support Affects and tuple results**	**Implementsemantic check and select statement filter**	**Extend Query Evaluator to support Affects and Affects***	**Extend Query Evaluator to support query tuple processing and projection**	**Write system test cases for the enhancement**
Adinda									
Lacie									
Hisyam									
Steven									
Ipsita									
Yohanes									

2. COMPONENTS

2.1. CODE PARSER

(Same as before)

2.2. PKB

2.2.1. DESIGN DECISIONS

In this iteration, the main focus of PKB is the data hiding, method simplifications and revamping the data structure.

Data hiding

Previously, other methods can get the individual classes inside PKB (such as Parent class) and utilise their method. This interaction has now been changed such that other classes can only use the methods that are provided in the PKB API. They will not be able to get the individual classes anymore.

Method Simplifications

Previously, there were many methods in PKB that were superfluous and outside of the scope of the PKB. This will not only introduce unnecessary bugs but also make it harder for the system to be optimised. As a result, a lot of methods have been trimmed down. Please refer to the PKB API to check the comprehensive methods list that PKB will now provide.

Revamping of Data Structure

The main focus of the revamping of data structure is speed. Therefore, PKB will now mainly use vector and bit array where the searching is mostly O(1).

PKB – Design Abstractions	
Tables	**Data Structures**
ProcTable	vector<PROCNAME> and map<PROCNAME,PROCINDEX>
ConstTable	vector<CONSTVALUE>
TypeTable	vector<SynType> and map<SynType,vector<STMTNUM>>
VarTable	vector< VARNAME > and map< VARNAME, VARINDEX >
Relationships	**Data Structures**
Follows	vector<STMTNUM>
Parent	vector<vector<int64_t>> and vector<STMTNUM>
Uses	vector<vector<int64_t>>
Modifies	vector<vector<int64_t>>
Calls	vector<vector<int64_t>>
Next	vector<vector<STMTNUM>> and vector<vector<pair<STMTNUM,STMTNUM>>>

For the tables, getting the value by index will be in O(1). Meanwhile, getting value by the name will be done in reverse mapping using map in O(log n). Map is used to avoid the worst case scenario of unordered_map which is O(n).

For the relationships, here are the design decisions.

1. **Follows**

   Follows will only need to record two statement numbers in each entry. One forward mapping and one reverse mapping are all that is required which can be achieved using vector. All search operations can be done in O(1).

2. **Parent**

   For the parent to children mapping, we will use bit array. This application of bit array is exploited with the fact that the children will always be after the parent. Therefore the bit array will store the number after the parent's statement number. In terms of storage, the number of int64_t that will be stored is dependent on = (last children statement number – parent statement number) / 63. In most cases, it will be mostly one which is space efficient. This application will make the searching of specific parent and children combination much faster at O(1). The speed of the rest of the operations will be the same if we were to use vector. In addition, the reverse mapping of children to parent is also provided using vector which can be done in O(1).

3. **Uses, Modifies and Calls**

   Uses, Modifies and Calls will all use bit arrays (and the reverse mapping as well). Using bit array will not only save memory but it will also speed up the searching compared to normal vector. Searching of a specific combination will be done in O(1) while listing down of all the values for a given index will be done in O(k) where k is the size of the answer.

4. **Next**

   For next, we will be using vector. We will also be using a separate table that stores a pair of ranges which is optimised for the Query Processor.

2.3. Design Extractor

The Design Extractor extracts the nextPair relationships using the Next relationships stored in PKB. It then stores these nextPair relationships in the PKB for future use by the Query Evaluator. By implementing a nextPair relationship, we are able to do the NextStar query faster in the Query Evaluator, without having to call the getNext() relationship multiple times when the program lines are sequentially connected in the CFG.

Extracting "NextPair" relationship

NextPair relationship is defined as a follows:
getNextPair() should return a vector of pairs where each pair denotes a range of program lines. So if getNextPair(x) returns [(a, b), (c, d)], then there is a path in the CFG from x to any program line between a and b (inclusive). There is also a path in the CFG from x to any program line between c and d (inclusive).

E.g. if the Next table is as follows:
…
3 → 4
4 → 5, 7
5 → 6
6 → 9
…

Then, the following nextPair relationships should be returned.
getNext(3): [(4, 6)]
getNext(4): [(5, 6), (7, 12)]
getNext(5): [(6, 6)]
getNext(6): [(9, 12)]

2.4. Query Processor

Query Processor consists of three parts: Query Processor (controller), Query Parser, and Query Evaluator.

2.4.1. Query Processor

Query Processor is a façade class for the whole component. Its responsibilities include:

1. Query Processor calls QueryParser to create a Query object from the given query string.
2. Query Processor then passes the Query object to the QueryEvaluator.
3. Query Evaluator will compute all necessary relations and return the results in the form of a list of integers.
4. Query Processor transforms the result into the correct display format and returns the answer to the user.

2.4.2. Query Parser

Query Validation

//no change from iteration 2 report

….
assign a; while w; Select a such that Follows(w, a) pattern a (“x”, _”x+y”_)
….

Query Parsing

The parser processes the *selectStatement* vector from the earlier. The *selectStatement* vector will be processed to construct a Query object with the following structure.

Query	
vector<string>	selected-synonym
vector<Relationship>	relationship-vector
map<string, synonym-type>	synonym-table

Since each query can contain many select clauses, the selected synonyms and clauses are stored inside a vector for scalability purposes. All the synonyms present in the *selected-synonym* vector will be detected and validated once again on whether they have been declared earlier. If it is not declared, the query is invalid and **QueryParser** will indicate and return invalid. The string vector will then be stored in a **Query** class.

The synonym map that was created earlier will also be included in the **Query** object. Both such-that and pattern clauses will be stored as another object, **Relationship**, as the following.

Relationship	
enum	relationship-type
enum	token-type
string	token1
string	token2
TokenType	token1-type
TokenType	token2-type
string	pattern-synonym

Relationship arguments will also be semantically checked to determine if they are valid. For example, both arguments in Follows clause have to be a statement (i.e. stmt, while, if, assign, call). If the arguments contain a constant synonym (e.g. constanct c; while w; Select w such that Follows (c,w), then **QueryParser** will detect the error and return invalid.

The types of the tokens/arguments are detected by **QueryParser** and store them in the **Relationship** class. For example, if the token is "Second", then its type will be IDENTIFIER, if the token is 1, then its type will be INTEGER, and if the token is w, then its type will be SYNONYM.

From the valid example above, the select-statement vector will be processed to produce the following.

Query	
selected-synonym	a
relationships	[rel1, rel2]
synonym-table	map1

rel1	
relationship-type	FOLLOWS
token1	w
token2	a
token1-type	SYNONYM
token2-type	SYNONYM

rel2	
relationship-type	PATTERN
token1	"x"
token2	_"x+y"_
token1-type	IDENTIFIER
token2-type	UNDERSCOREEXPR
pattern-synonym	a

map	
Synonym	Type
a	ASSIGN
w	WHILE
BOOLEAN	BOOLEAN

When the controller calls the parsing function, the function will return a query object. This object will then be passed to query evaluator.

For with-clause, Query Parser with detect the conditions whether they are valid.

For example:

procedure p,q; Select q such that Calls (p,q) with p.procName=”Second”

p.procName = ”Second” will be parsed into two parts, left-hand-side and right-hand-side. Right-hand-side includes “Second” and will be stored in token2/argument-2 in a relationship object. Left-hand-side includes p.procName and will be stored in token1/argument-1 if token is valid. Synonym p will be checked against the map whether it exists. Since the attribute name is procName, it will also be checked on whether it is of type procedure.

Left-hand-side and right-hand-side will be checked on whether they are of the same type, either character strings or integers. If they are of different type, return invalid.

If it passes both validations, then the query is valid and stored in the relationship below:

rel2	
relationship-type	WITH
token1	p
token2	“Second”
token1-type	SYNONYM
token2-type	IDENTIFIER

“procName” is not stored because it is known that a synonym of type procedure can only have “procName” as its attribute name.

For multiple clauses (such that, pattern, with), Query Parser works by keeping the previous clause keyword (e.g. “pattern”, or “with”), and use it to detect the clause type when it encounters “and”.

For example:

assign a; Select a pattern a(_,_) and a(“x”,_”x+y”_);

Parser will detect the first clause and store in the relationship class accordingly

rel1	
relationship-type	PATTERN
token1	_
token2	_
token1-type	UNDERSCORE
token2-type	UNDERSCORE
pattern-synonym	a

It also keeps track of the last relationship-type. Therefore, when Query Parser reaches the word “and”, it knows that it will be parsing a pattern clause again, translating the “and” keyword into “pattern”. Query Parser will then validate and parse accordingly with respect of the clause type, in this case it is “pattern”.

rel2	
relationship-type	PATTERN
token1	“x”
token2	_”x+y”_
token1-type	IDENTIFIER
token2-type	UNDERSCOREEXPR
pattern-synonym	a

Both relationships will then be stored in a **Query** class and passed to **QueryEvaluator**.

2.4.3. Query Evaluator

(same as before)

3. TESTING

3.1. SYSTEM TESTING PLAN FOR ITERATION 3

The following tables illustrate the system test cases used during Iteration 3

Level	Function	Parameters	Select
Basic	Affects	#, a2	a2
		#, a2	boolean
		a1, #	a1
		a1, #	boolean
		#, #	boolean
		a1, a2	a1
		a1, a2	a2

Level	Function	Parameters	Select
Basic	Affects*	#, a2	a2
		#, a2	boolean
		a1, #	a1
		a1, #	boolean
		#, #	boolean
		a1, a2	a1
		a1, a2	a2

Level	Function	Parameters	Select
Intermediate	Tuple	Calls*(p1, p2)	<p1, p2>
		Calls(p1, p2)	<p2, p1>
		Modifies*(s1, s2)	<s1, s2>
		Modifies(s1, s2)	<s2, s1>
		Modifies*(p1, s1)	<p1, s1>
		Modifies(p1, s1)	<s1, p1>
		Uses*(s1, s2)	<s1, s2>
		Uses(s1, s2)	<s2, s1>
		Uses*(p1, s1)	<p1, s1>
		Uses(p1, s1)	<s1, p1>
		Parent*(s1, s2)	<s1, s2>
		Parent(s1, s2)	<s2, s1>
		Follows*(s1, s2)	<s1, s2>
		Follows(s1, s2)	<s2, s1>

		Next*(n1, n2)	<n1, n2>
		Next(n1, n2)	<n2, n1>
		Affects*(a1, a2)	<a1, a2>
		Affects(a1, a2)	<a2, a1>

Level	Function	Clause
Intermediate	Multiple clause using Affects	Modifies*
		Uses*
		Parent*
		With

Level	Function	Clause
Intermediate	Multiple clause using Affects*	Follows*
		Affects
		With

Level	Function	Clause 1	Clause 2	Select
Advanced	Multiple clause using tuple	Calls*(p1, p2)	Modifies*(p1, s1)	<p1, p2>
		Calls*(p1, p2)	Modifies*(p1, s1)	<p1, s1>
		Modifies*(p1, s1)	Uses*(s1, s2)	<p1, s2>
		Uses*(p1, s1)	Parent*(s1, s2)	<s1, s2>
		Next(n1, n2)	Next*(n2, n3)	<n1, n3>
		Follows*(a1, a2)	Affects*(a1, a2)	<a1, a2>
		Calls*(p1, p2)	Uses*(s1, s2)	<p1, s2>

4. API

Please view our Doxygen at:

www.comp.nus.edu.sg/~kester/cs3202