National University of Singapore
School of Computing

CS3202: Software Engineering Project II
TEAM 05: Flying Cockroach

Semester 1, AY2014/2015

Date of Submission: 13 Nov 2014

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 6-6.30pm

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 SPA during CS3201 and CS3202.

1.1. ARCHITECTURE

The architecture for the prototype consists of 3 main components: the Code Parser, the PKB and the Query Processor. Both the Code Parser and the Query Processor are dependent on PKB but not dependent on each other. 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.

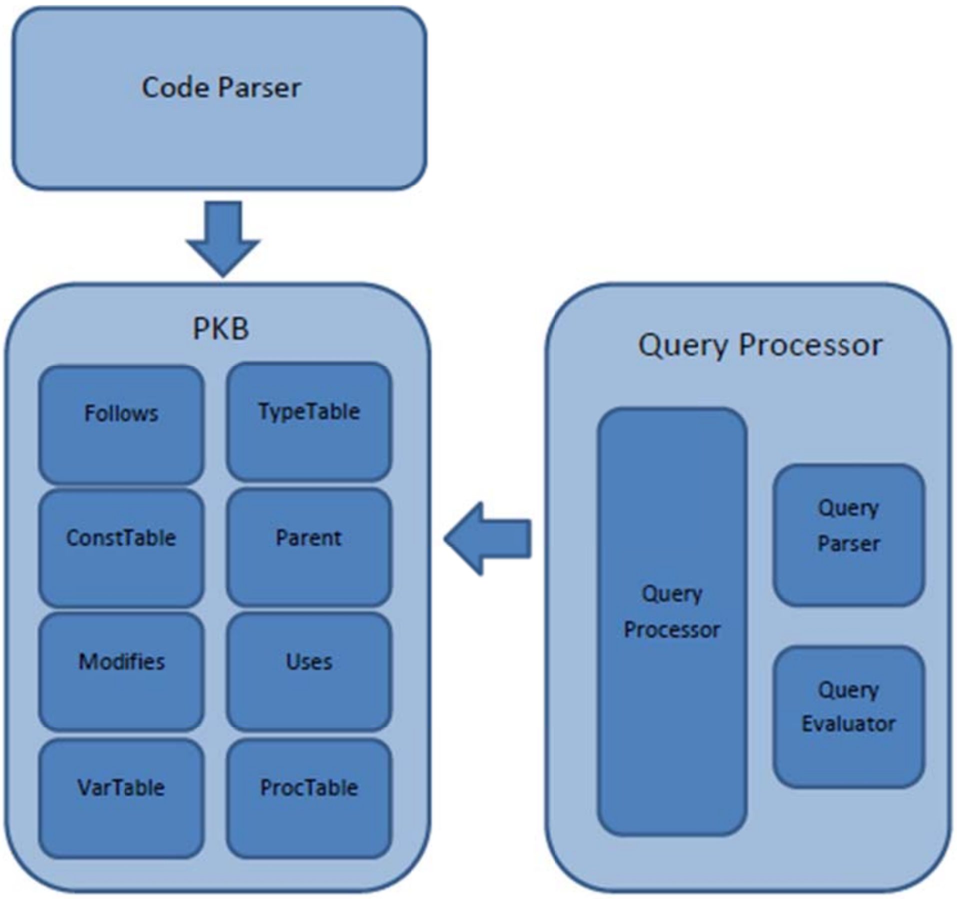


Figure 1

1.2. INTERACTION

CodeParser works by evaluating each line of the given source code. It creates AST Node, sets the pointers accordingly; sets the tables and the appropriate databases in PKB.

The attributes in PKB (the tables) will then be used by Query evaluator to answer queries. Testing for CodeParser is done by checking the content of each table, whether it has set the values properly, and check the content of each node in the AST, whether it matches the correct AST.

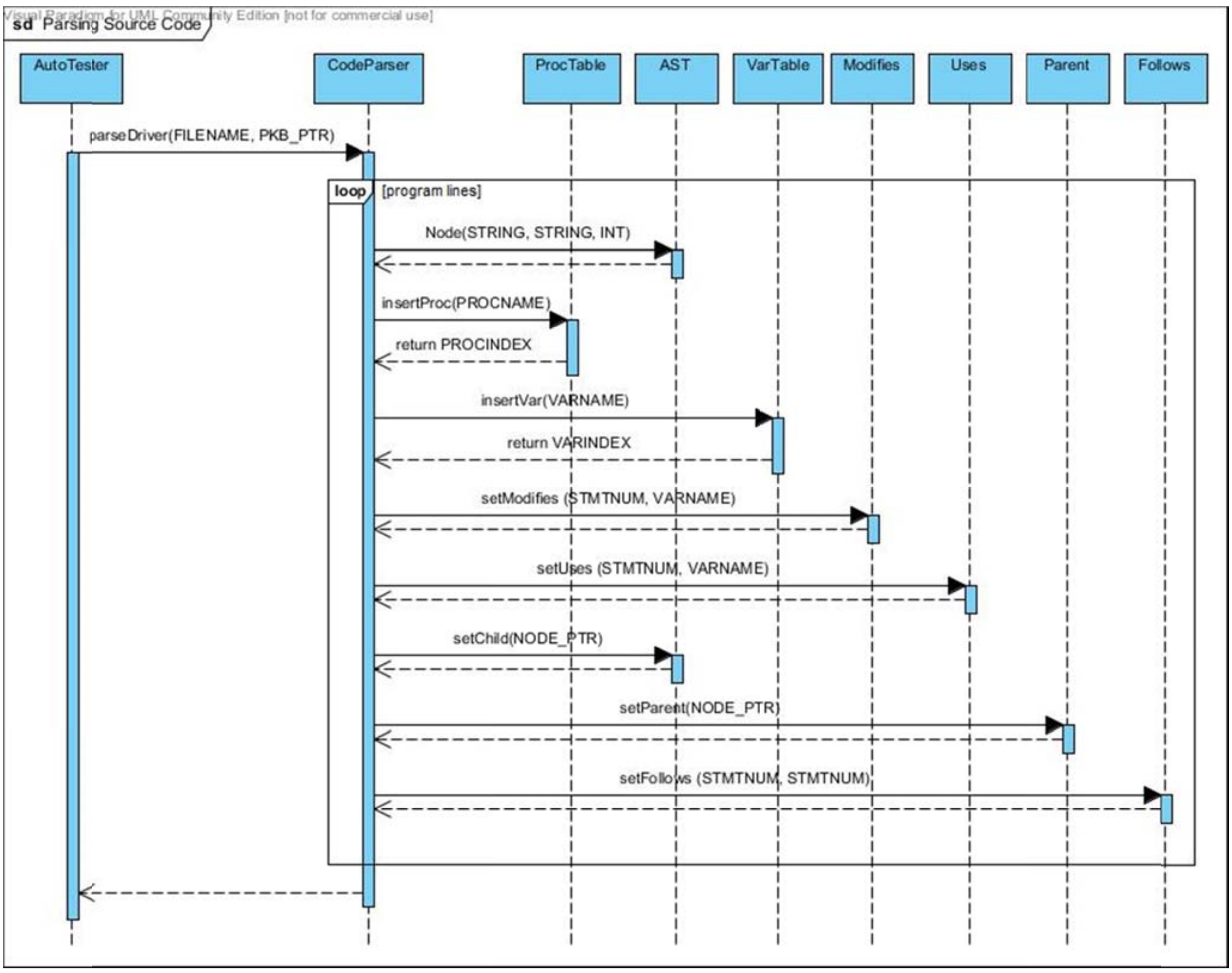


Figure 2

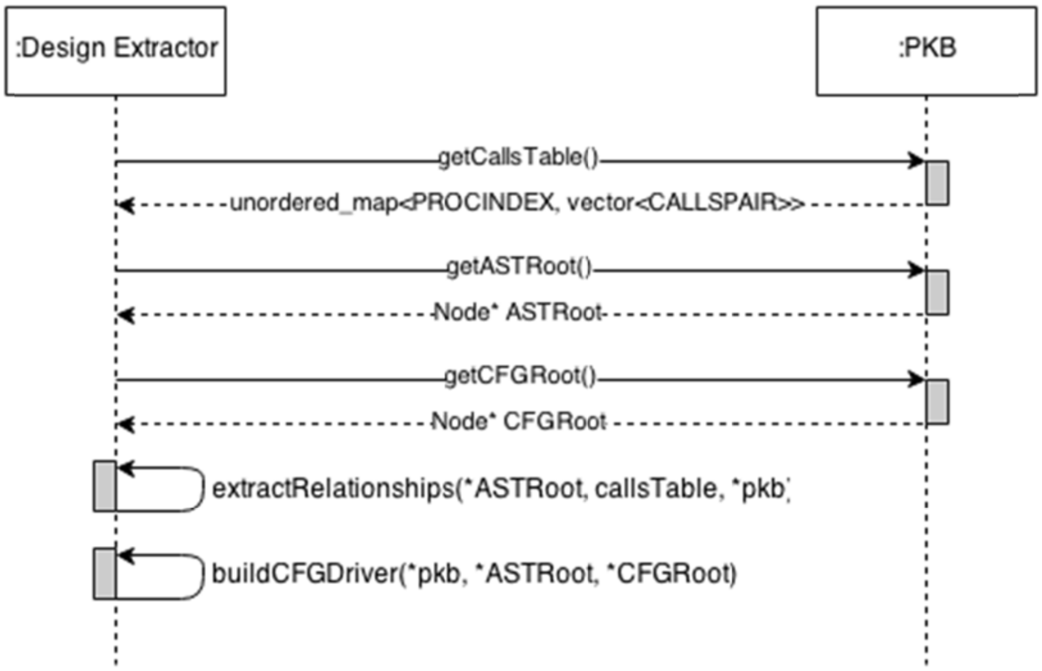

:Design Extractor
:PKB
getCallsTable()
unordered_map<PROCINDEX, vector<CALLSPAIR>>
getASTRoot()
Node* ASTRoot
getCFGRoot()
Node* CFGRoot
extractRelationships(*ASTRoot, callsTable, *pkb)
buildCFGDriver(*pkb, *ASTRoot, *CFGRoot)

Figure 3

Figure 3 shows the interaction of the Design Extractor with the PKB. It is in charge of extracting other relationships that cannot be extracted from the SIMPLE program during the preliminary parsing phase. After extracting the modifies and uses relationships using the callsTable supplied by the PKB, Design Extractor stores these additional relationships (such as modifies and uses for calls statements and for container statements) into the PKB. After that the Design Extractor builds the CFG using the AST root provided by the PKB. It then stores the pointer to the CFG that it has created back into the PKB.

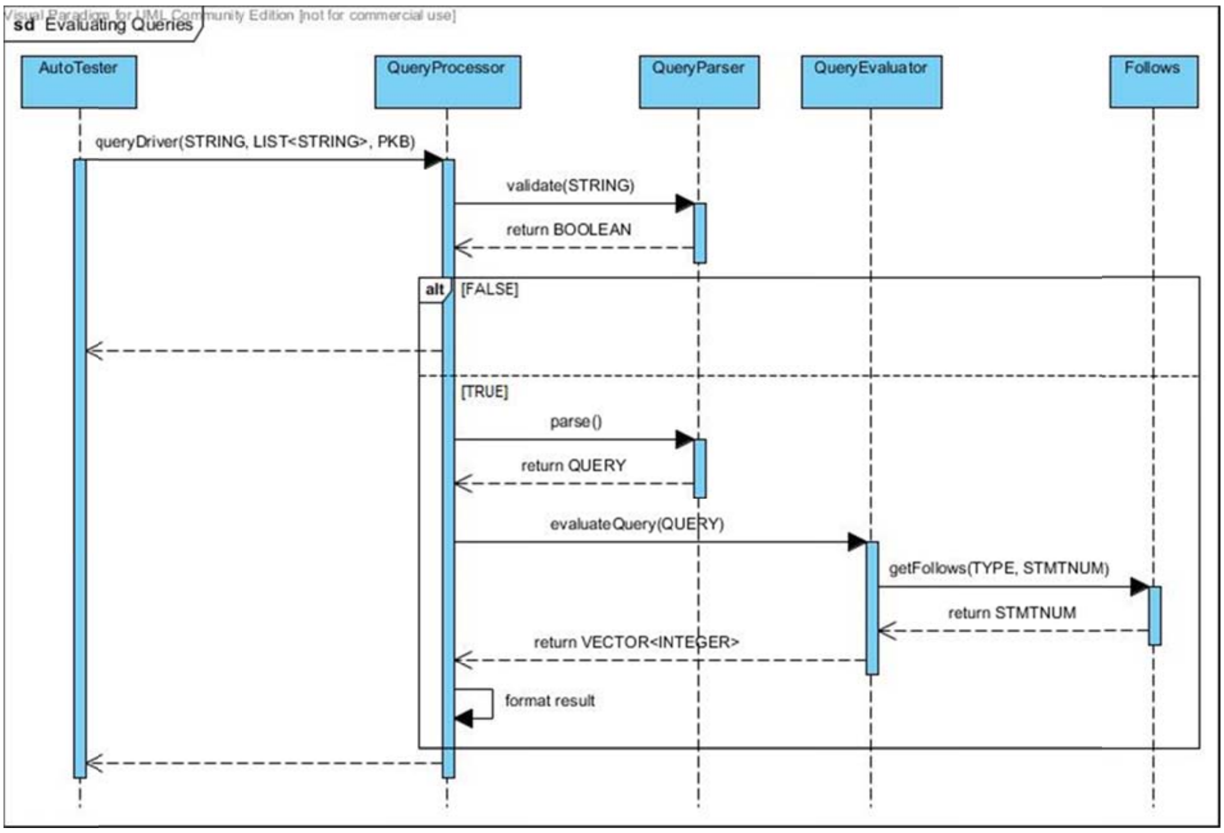


Figure 4

Figure 4 shows the sequence diagram of the query evaluation process. This diagram was useful in demarcating the responsibilities of each PQL group member. For example, QueryEvaluator directly assumes that the Query it receives is valid and syntactically correct. Therefore it is the responsibility of QueryParser to validate each query before passing it to the evaluator.

This diagram also helps to keep track of the dependencies between components. This is especially useful during debugging process of integration testing. When QueryProcessor fails to return the correct result, the team knows that the errors could come from at least three places, i.e. QueryParser, QueryEvaluator, and PKB.

2. SUMMARY OF ACHIEVEMENTS

2.1. BASIC SPA

All the required functionalities for Iteration 1-3 were implemented. These include:

1. All the grammar rules for the SIMPLE source program as outlined on Page 103 of the Handbook.
2. All the grammar rules for the PQL as described on Page 98-101 of the Handbook.
3. All the relationships are extracted and can be queried using PQL. These include:

Relationship	**Implemented using**
Calls and Calls*	CodeParser, PKB, Query Evaluator
Modifies	CodeParser, Design Extractor, PKB, Query Evaluator
Uses	CodeParser, Design Extractor, PKB, Query Evaluator
Parent and Parent*	CodeParser, PKB, Query Evaluator
Follows and Follows*	CodeParser, PKB, Query Evaluator
Next and Next*	DesignExtractor, PKB, Query Evaluator
Affects and Affects*	PKB, Query Evaluator

2.2. Bonus Features

We have extended the features of SPA beyond its call of duty. The features will be explained in more detail below:

2.2.1. Sibling Relationship

We implemented the Sibling relationship as suggested by the Assignment 4 specifications.

2.2.2. Flexible CodeParser

```
procedure Orchid {
      while idx {
             y = z*3 + 2*x;
             call Tulip;
             idx = idx - 1; }
      z = z + y + idx; }

procedure Lily {
      while v {
             y = x + y; }
      x = y + x; }
```

According to the handbook, the source code that is to be tested against our SPA, is defined to be in standardized format and neatly arranged. They have regular and consistent spacing, indentation, tabs, and endline characters. Please refer to the example below:

However, our SPA source code parser is very flexible. It is able to detect erroneous and inconsistent spaces, tabs, endline characters and erroneous close curly brackets. A flexible CodeParser is, thus, one of our bonus features, as it was not required for us to have in the

```
procedure           Orchid


{
while      idx    {

             y = z      *3 + 2*   x    ;
             call       Tulip    ;

             idx    =    idx            -1;

}

                    z=z+y+idx;
}

procedure Lily{
                          while v{

                    y           =x              + y            ;}
x = y + x;}
```

basic SPA. To demonstrate this ability, please refer to the example below (which will have the same extracted information as the code fragment above):

```
if ifstat; Select ifstat such that Follows* (ifstat, 17)

assign a; Select a such that Modifies (a, "idx") and Uses (a, "idx")

assign a; while w; Select a such that Modifies (a, "idx") and Uses (a, "idx") and
Follows (15, a) and Parent* (w, a)
```

2.2.3. Flexible QueryParser

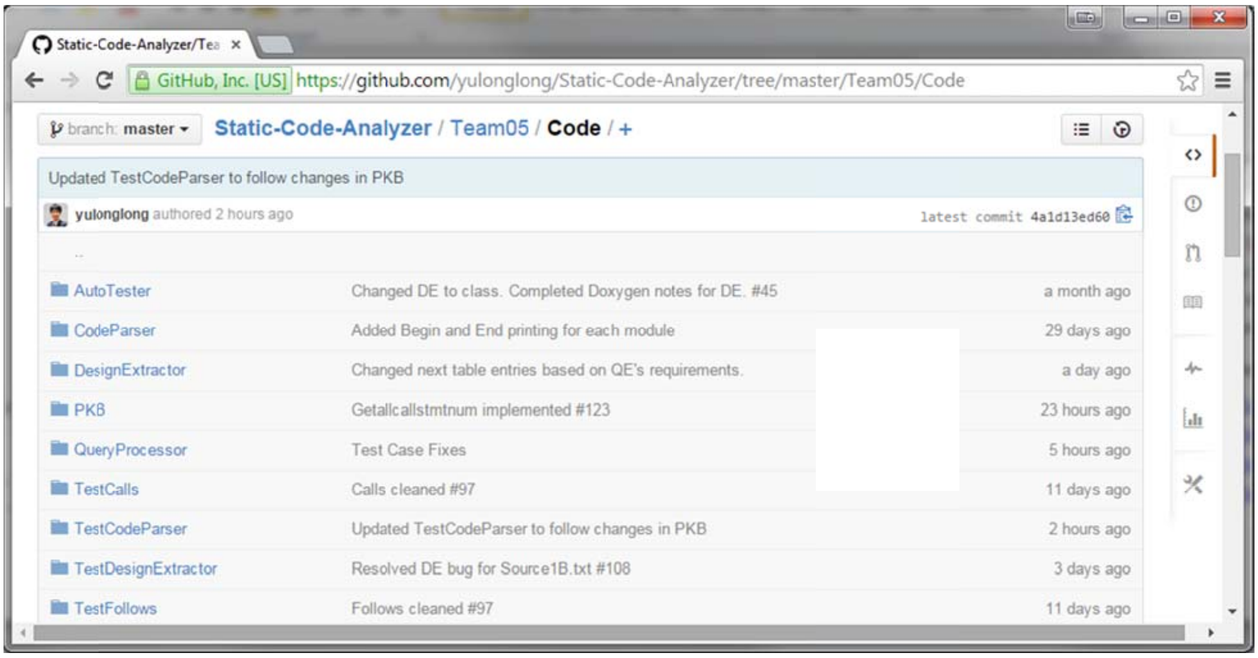
The queries that are to be tested against basic SPA are defined to be in a standard format. They have regular and consistent lowercase/uppercase command, spacing, and characters. Please refer to the sample standard query below:

```
If    ifstat    ; Select     ifstat    such   that   Follows*   (   ifstat,17)

assign a;Select a such that Modifies(a,"idx") and Uses(a,"idx")

assign     a   ;    while   w    ; Select     a such that Modifies (a,   "idx"   ) and
Uses    (   a  ,    "idx")     and       Follows      (15, a)    and    Parent*
(   w   ,   a   )
```

However, our QueryParser is very flexible and is able to detect queries with inconsistent/extra spaces and lowercase/uppercase clauses. As it was not required for us to have such flexibility for CS3202's basic SPA, we have this advanced feature. To demonstrate this ability, please refer to the example below (which will have the same information/result as the example above):

2.2.4. Highly Organised Repository

In our shared repository, our code, report files and directories are very well organised. We do this by following standard naming conventions and following a structured hierarchy such that each team member has fast and easy access.



Static-Code-Analyzer/Tea ×

GitHub, Inc. [US] https://github.com/yulonglong/Static-Code-Analyzer/tree/master/Team05/Code

branch: master ▾ Static-Code-Analyzer / Team05 / **Code** / +

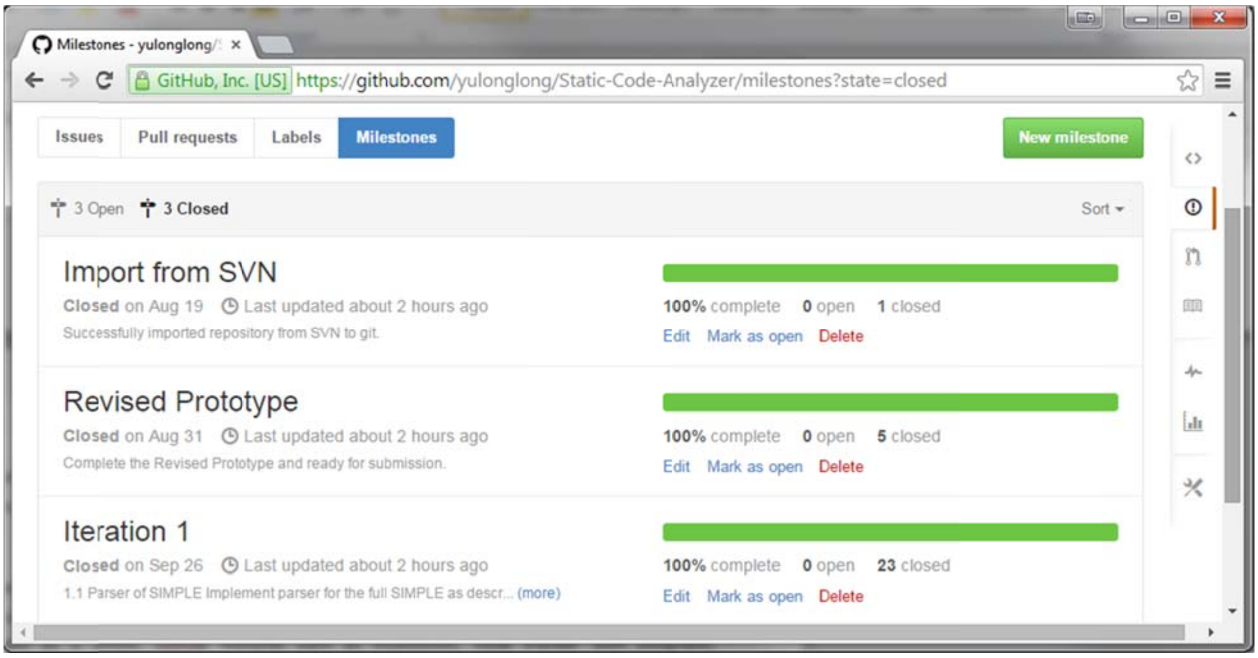
Updated TestCodeParser to follow changes in PKB

yulonglong authored 2 hours ago latest commit 4a1d13ed60

..

AutoTester	Changed DE to class. Completed Doxygen notes for DE. #45	a month ago
CodeParser	Added Begin and End printing for each module	29 days ago
DesignExtractor	Changed next table entries based on QE's requirements.	a day ago
PKB	Getallcallstmtnum implemented #123	23 hours ago
QueryProcessor	Test Case Fixes	5 hours ago
TestCalls	Calls cleaned #97	11 days ago
TestCodeParser	Updated TestCodeParser to follow changes in PKB	2 hours ago
TestDesignExtractor	Resolved DE bug for Source1B.txt #108	3 days ago
TestFollows	Follows cleaned #97	11 days ago

Appropriate milestones and issues are also tracked and reported regularly to achieve any goals or/and objectives.



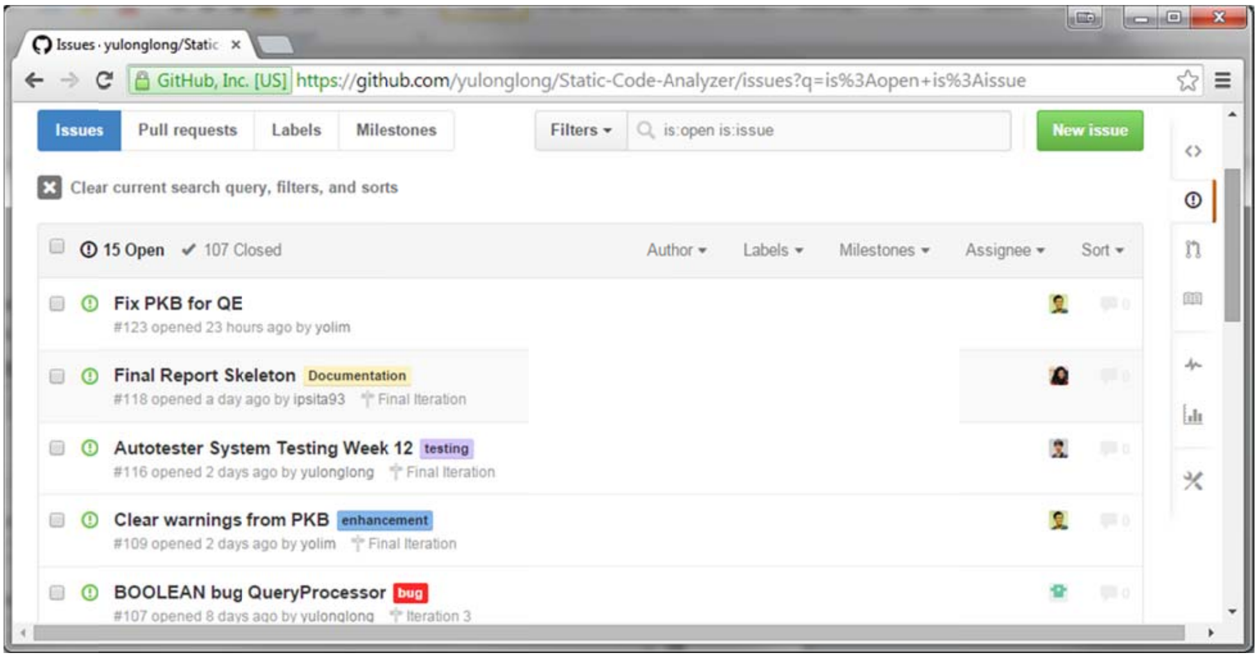
Milestones - yulonglong/ ×

GitHub, Inc. [US] https://github.com/yulonglong/Static-Code-Analyzer/milestones?state=closed

Import from SVN
Closed on Aug 19 Last updated about 2 hours ago
Successfully imported repository from SVN to git.
100% complete 0 open 1 closed
Edit Mark as open Delete

Revised Prototype
Closed on Aug 31 Last updated about 2 hours ago
Complete the Revised Prototype and ready for submission.
100% complete 0 open 5 closed
Edit Mark as open Delete

Iteration 1
Closed on Sep 26 Last updated about 2 hours ago
1.1 Parser of SIMPLE Implement parser for the full SIMPLE as descr... (more)
100% complete 0 open 23 closed
Edit Mark as open Delete


Issues · yulonglong/Static

GitHub, Inc. [US] https://github.com/yulonglong/Static-Code-Analyzer/issues?q=is%3Aopen+is%3Aissue

Issues | Pull requests | Labels | Milestones | Filters | is:open is:issue | New issue

Clear current search query, filters, and sorts

15 Open ✓ 107 Closed — Author ▾ Labels ▾ Milestones ▾ Assignee ▾ Sort ▾

Fix PKB for QE
#123 opened 23 hours ago by yolim

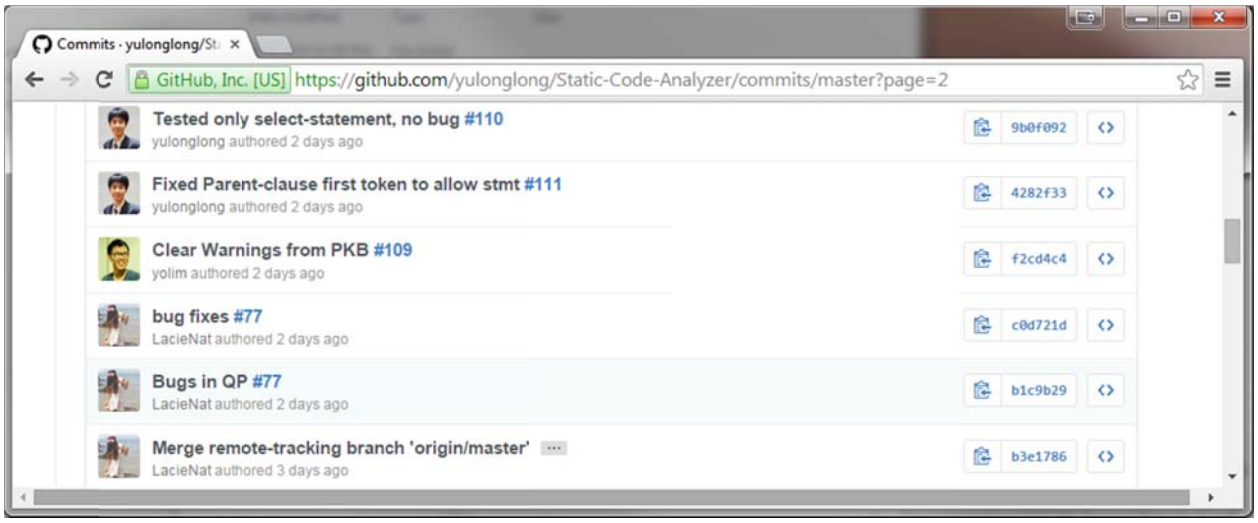
Final Report Skeleton `Documentation`
#118 opened a day ago by ipsita93 Final Iteration

Autotester System Testing Week 12 `testing`
#116 opened 2 days ago by yulonglong Final Iteration

Clear warnings from PKB `enhancement`
#109 opened 2 days ago by yolim Final Iteration

BOOLEAN bug QueryProcessor `bug`
#107 opened 8 days ago by yulonglong Iteration 3

Every commit related to certain issues or milestones was tagged and linked to the issue for easy tracking and reference in the future. It was also used to monitor the progress of any issues and the SPA system as a whole. Github features such as milestones, issue tracker with assignees, milestones, and issue category (bugs, testing, documentation, enhancement, etc) were highly useful for our project management process.


Commits · yulonglong/St

GitHub, Inc. [US] https://github.com/yulonglong/Static-Code-Analyzer/commits/master?page=2

Tested only select-statement, no bug #110
yulonglong authored 2 days ago — 9b0f092

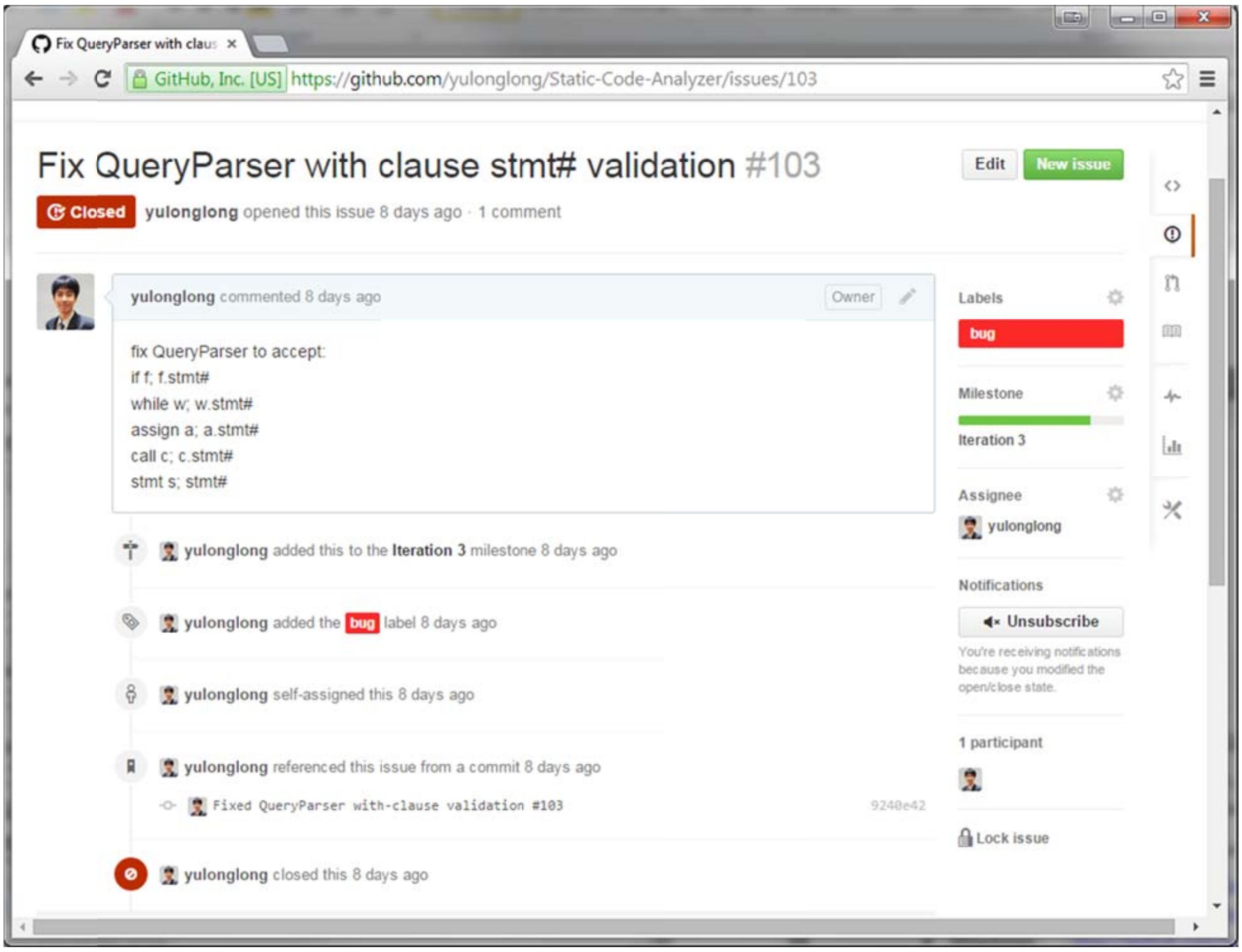
Fixed Parent-clause first token to allow stmt #111
yulonglong authored 2 days ago — 4282f33

Clear Warnings from PKB #109
yolim authored 2 days ago — f2cd4c4

bug fixes #77
LacieNat authored 2 days ago — c0d721d

Bugs in QP #77
LacieNat authored 2 days ago — b1c9b29

Merge remote-tracking branch 'origin/master' …
LacieNat authored 3 days ago — b3e1786



Fix QueryParser with clause stmt# validation #103

Closed yulonglong opened this issue 8 days ago · 1 comment

yulonglong commented 8 days ago

```
fix QueryParser to accept:
if f; f.stmt#
while w; w.stmt#
assign a; a.stmt#
call c; c.stmt#
stmt s; stmt#
```

yulonglong added this to the Iteration 3 milestone 8 days ago

yulonglong added the bug label 8 days ago

yulonglong self-assigned this 8 days ago

yulonglong referenced this issue from a commit 8 days ago

`Fixed QueryParser with-clause validation #103` 9240e42

yulonglong closed this 8 days ago

Labels: bug

Milestone: Iteration 3

Assignee: yulonglong

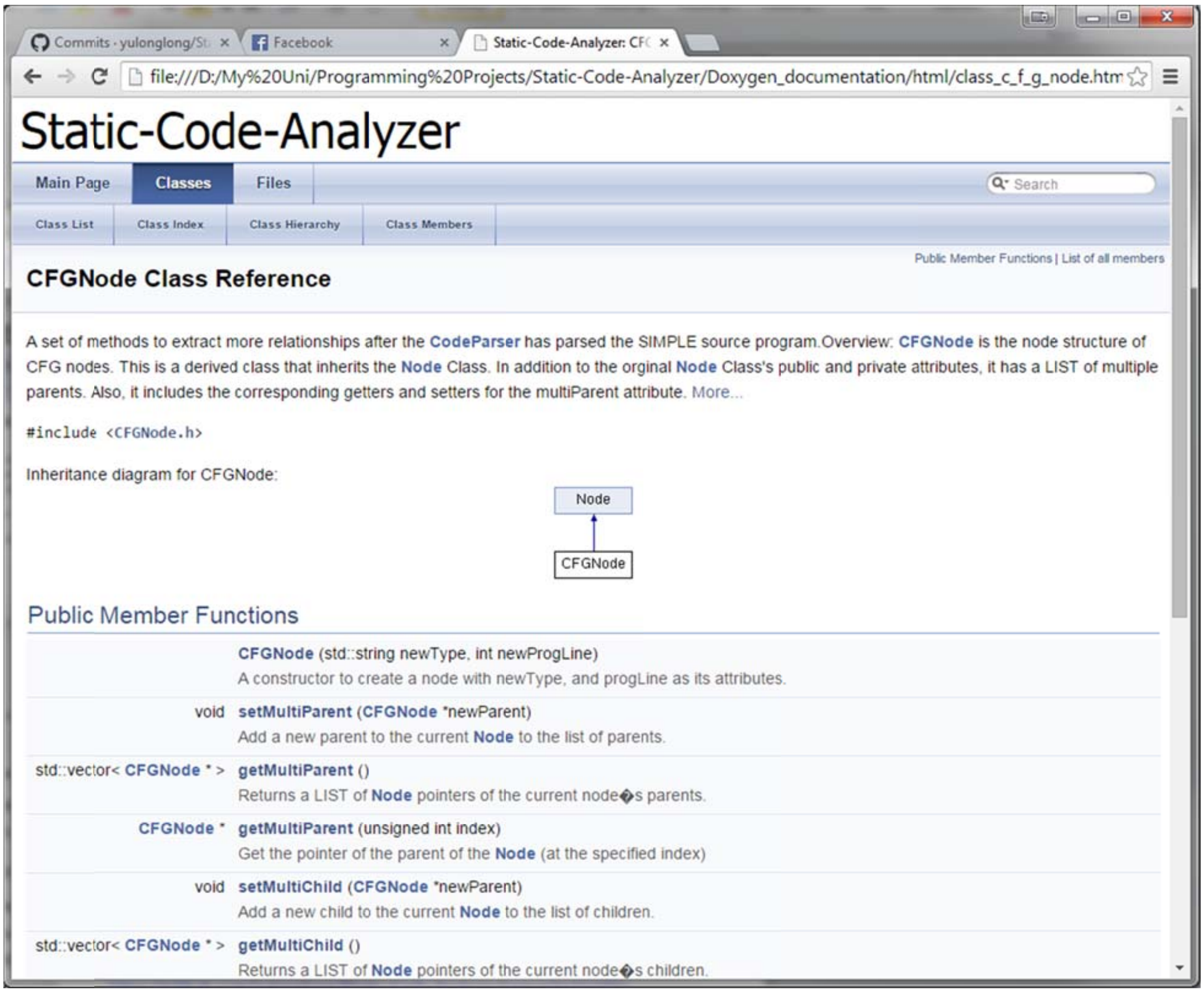
Notifications: Unsubscribe

You're receiving notifications because you modified the open/close state.

1 participant

Fix QueryParser with claus
GitHub, Inc. [US] https://github.com/yulonglong/Static-Code-Analyzer/issues/103
Edit
New issue
Owner
Lock issue

Overall, our code and API are well-documented and in sync with each other. Every detail of the code is described clearly, showing and explaining all methods, attributes, and inheritance diagram whenever applicable.



Commits - yulonglong/St
Facebook
Static-Code-Analyzer: CF
file:///D:/My%20Uni/Programming%20Projects/Static-Code-Analyzer/Doxygen_documentation/html/class_c_f_g_node.htm
Main Page
Classes
Files
Search
Class List
Class Index
Class Hierarchy
Class Members
Public Member Functions | List of all members
Node
CFGNode

Static-Code-Analyzer

CFGNode Class Reference

A set of methods to extract more relationships after the **CodeParser** has parsed the SIMPLE source program.Overview: **CFGNode** is the node structure of CFG nodes. This is a derived class that inherits the **Node** Class. In addition to the orginal **Node** Class's public and private attributes, it has a LIST of multiple parents. Also, it includes the corresponding getters and setters for the multiParent attribute. More...

```
#include <CFGNode.h>
```

Inheritance diagram for CFGNode:

Public Member Functions

	CFGNode (std::string newType, int newProgLine) A constructor to create a node with newType, and progLine as its attributes.
void	**setMultiParent** (**CFGNode** *newParent) Add a new parent to the current **Node** to the list of parents.
std::vector< **CFGNode** * >	**getMultiParent** () Returns a LIST of **Node** pointers of the current node�s parents.
CFGNode *	**getMultiParent** (unsigned int index) Get the pointer of the parent of the **Node** (at the specified index)
void	**setMultiChild** (**CFGNode** *newParent) Add a new child to the current **Node** to the list of children.
std::vector< **CFGNode** * >	**getMultiChild** () Returns a LIST of **Node** pointers of the current node�s children.

Overall, our progress in building this project was conscientiously and thoroughly tracked and documented so that any new addition to the team could understand the project. Also, by posting and tracking issues, we can determine who tackled what.

Our project is currently hosted on GitHub, and the relevant links are as follows:

- Repository : https://github.com/yulonglong/Static-Code-Analyzer
- Issue tracker : https://github.com/yulonglong/Static-Code-Analyzer/issues
- Milestones : https://github.com/yulonglong/Static-Code-Analyzer/milestones
- Commits : https://github.com/yulonglong/Static-Code-Analyzer/commits/master

3. Project Plan

3.1. Schedule For Whole Project

Iteration	Major Task	Target Completion	Completed
0	Change query parser to use regular expression checking	Week 3	Week 2
0	Improve internal PKB data structures	Week 3	Week 2
1	Implement Calls, Modifies, and Uses in PKB	Week 5	Week 5
1	Extend Query Parser to support with clause	Week 5	Week 5
1	Extend Query Evaluator to support with clause	Week 5	Week 6
2	Implement Next in PKB	Week 7	Week 7
2	Extend Query Parser to support multiple clauses	Week 7	Week 7
2	Extend Query Evaluator to support multiple clauses	Week 7	Week 8
3	Implement Affects in PKB	Week 10	Week 10
3	Extend Query Parser to support Affects and tuple results	Week 10	Week 10
3	Extend Query Evaluator to support Affects and tuple result	Week 10	Week 12
4	Implement Sibling in PKB	Week 12	Week 12
4	Extend Query Parser to support Sibling	Week 12	Week 12
4	Extend Query Evaluator to support Sibling	Week 12	Week 12

3.2. Comments & Problems

Due to some difficulties, some of project milestones had to be delayed by a week. The following are the causes of the delay and the approach that the team took to circumvent them.

3.2.1. Difficulties in inter-module communications

The team found it difficult to communicate complex ideas, especially pertaining to implementation of concepts in the later iterations, e.g. multiple clause and Affects. Because of this, some components took longer to complete due to miscommunication, and conflicting interpretation of the implementation strategies.

The team then realized that it is crucial to create the API and .h files of major components in advance, to aid the communication process between team members. The API set the rules and the .h files will prevent compilation errors when the completion time of each module's development is asynchronous.

3.2.2. Lack of debug workflow

At first, the team did not have a debug workflow that could help identify where a bug was coming from, during system testing. This created confusion every time a test case failed, and prevented a fix to be effectively delivered.

Knowing this, the team utilized a debug mode during system testing that flags every checkpoint of the program, typically at the beginning and end of a module. This way, the team could quickly identify which module caused the bug, and communicated this bug to the team member in charge. The presence of this debug mode streamlined the development workflow and shortened the testing cycle timeline.

3.2.3. Difficulties in coordination

The team found it hard to coordinate tasks among team members with different schedules and priorities. The difficulty to find common available time for group meeting and discussion created a bottleneck for team communication. Progress checking among team members also became unclear because it was only facilitated through online media.

The solution to this problem was that team members put more effort in carrying out frequent discussions, albeit separately. Arranging for phone calls, or face-to-face meeting, when a problem is encountered was important in keeping the communication flow healthy.

4. Components

4.1. Code Parser

Code Parser's main functions are to read in the source code, build the AST, and set the tables (VarTable, ProcTable, TypeTable, Follows, Parent, Modifies, Uses) in PKB according to the input source code.

To build the AST, Code Parser depends on the implementation of node, which is the node structure being used to build AST. A node can have many children, thus we decided to use NODE_PTR_LIST (C++ vector) for dynamic storage of the children pointers, rather than array with fixed size.

Code Parser keeps track of every relevant parent of each node being built, by storing pointers to their parents. For example when there is a while statement, a pointer to the while statement will be stored and then link all the children to their parent node.

Code Parser works by tokenizing the source code as string line by line, detecting the tokens in each line, generating the types, program lines, and setting the tables accordingly.

For an assignment statement, Code Parser will convert the expression from infix to postfix, and then create the expression tree.

Code Parser does its validation by keeping track of the curly brackets (i.e. "{" and "}"). It keeps track of the number currently present open curly bracket, "{". When Code Parser encounters an open curly bracket, it will push it to a stack. When it encounters a closed curly bracket, it will pop from the stack.

When Code Parser reaches the end of the source code, it will return invalid if the stack is not empty, or if Code Parser is trying to pop from an empty stack. It means there is a mismatch in the number of curly brackets.

When the stack is empty, Code Parser will accept a line which defines a procedure. If it encounters any other statement while the stack is empty, it will return invalid.

Example:

Source code (source1.txt):

```
procedure Mini {
A1 = 29;
a1 = 31;
i = 51; }
```

The following describes how the Code Parser parses the above SIMPLE program:

1. Code Parser starts reading the source code at line 1, it will check whether the stack is empty. If the stack is empty, it will be expecting a procedure declaration.
2. It then parses procedure Mini, creates an ASTNode, sets it as root, and pushes the curly bracket “{“ into the stack. Insert “Mini” into the ProcTable.
3. At line 2, Code Parser tokenizes and checks the type of statement. Since it starts with a variable, it detects the statement as an assignment statement.
4. Code Parser will check if there exists a semicolon at the end of the line because it is compulsory to have a semi colon at the end of an assignment statement. If it exists, it will create a node containing “=”, and link “A1” as the first child. The expression on the right hand side will be converted into a postfix expression, and then build the expression tree.
5. Code Parser will link the root of the expression tree as the second child of “=”, and then link the “=” to its parent, which in this case, is “procedure Mini”.
6. Code Parser will also set the VarTable, Modifies, Uses, Follows, and Parent accordingly, in this case, it will set Modifies (line 1, and variable A1), and insert A1 into the VarTable.
7. At line 3, it detects that it is an assignment statement, and repeat step 4 to 6.
8. At line 4, it detects that it is an assignment statement, and repeat step 4 to 6. In addition, Code Parser detects a closed curly bracket. Therefore, Code Parser will pop the stack.
9. End of source code is reached. It will now check whether the stack is empty.
10. Since the stack is empty, and there is no violation of the rules stated earlier. Code Parser has built the AST successfully and stored the design abstractions in the relevant tables.

CodeParser’s Unit testing is done by checking the content of each table, whether it has set the tables properly, and by checking the contents of each node in the AST, whether it matches the expected AST.

4.2. PKB

4.2.1. Design Decisions

PKB is implemented using the singleton pattern. One instance of PKB will be initialised during the construction phase of the UI (which is AutoTester). Afterwards, we will only pass the PKB pointer to other components which need to alter the PKB or call the PKB's methods. This is to ensure that other components are always editing or accessing the same PKB object. Using the same rationale, all the sub-components of PKB (VarTable, ProcTable, ConstTable, Follows, Parent, Modifies and Uses) are singleton classes and only their pointers are passed around.

To make things clearer during communication, we used some new definitions:

- Typedef int
  - STMTNUM: for statement number
  - VARINDEX: for variable index
  - CONSTINDEX: for constant index
  - PROCINDEX: for procedure index
- Typedef string
  - VARNAME: for variable name
  - PROCNAME: for procedure name
  - CONSTVALUE: for constant value
- Typedef Enum SynType { ASSIGN, IF, WHILE, STMT, BOOLEAN, CALL, VARIABLE, CONSTANT, PROGLINE, PROCEDURE, PLUS, MINUS, TIMES, INVALID, STMTLST} TYPE
  - to discern the type of each statement number
- Typedef pair<PROCINDEX,STMTNUM> CALLSPAIR;
  - To store the index of the procedure being called and the statement number where the call is invoked

The data structure used for the tables and the relationships is mainly vector as listed below.

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>>>

In deciding the data structures to be used in PKB, our main consideration was speed – in inserting and searching. In making our design decision, we made a comparison between vector, ordered map and unordered map.

	Vector of Size N	**Ordered Map of Size N**	**Unordered Map of Size N**
Insert	O(1) or O(N) if need resizing	O(log N)	O(1) (Average case) O(N) (Worst case)
Search	O(1)	O(log N)	O(1) (Average case) O(N) (Worst case)

We can see that vector is the fastest of the three which explains our decision to use it.

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.

4.2.2. Interaction with other Components

The PKB mainly interacts with the Code Parser and the Query Evaluator.

Interaction with Code Parser

After PKB is initialised, the PKB pointer is passed to Code Parser to fill in the tables and relationships into the PKB.

For example, when the Code Parser calls insertVar(“x”), the following is done:

1. Check whether the given variable, “x”, exists in the table by iterating through the element in the table one by one.
2. If yes, we will simply just return the index.
3. Otherwise, insert the element at the back of the table and return the index (table size - 1) of the variable.

For example, when the Code Parser calls setUses(12,”x”)the following is done:

1. Get the variable index of “x” from VarTable.
2. If the variable index is -1, it means that there is no such variable and thus the method will terminate.
3. If the variable index is more than -1, it means that the variable exists, therefore insert into the uses table at key 12, the value of variable index “x”.

Interaction with Design Extractor

The PKB pointer is then passed to Design Extractor to extract more design abstractions and relationships that were not picked up by the CodeParser during the parsing stage. In short, the Design Extractor sets Modifies and Uses relationships for procedures and calls statements as well as the Next and Sibling relationship.

Interaction with Query Evaluator
PKB pointer is then passed to Query Processor so that Query Evaluator can call the public API provided by PKB. Query Processor will need to get the tables or relationships that it needs first and only then it can call the corresponding API that it needs.

For example, if we want to call getChildren(WHILE, CALL), Query Processor needs to get parent from PKB and then calls parent->getChildren(WHILE, CALL). When it is called, it will result a list of STMTNUM x such that for each x, Parent(CALL, x) holds and x is a WHILE statement. If there exists no such statement x, an error code is returned. The steps are as follows:

1. Get parent pointer from PKB using getParent()
2. Calls the method getChildren(WHILE,CALL) from parent
3. Iterate the children table inside parent from beginning to end. The index of the vector, i, will indicate the statement number of the children.
4. Get j, the value of the vector at the specified index which is the statement number of the parent
5. If j is -1, continue with the next index from step 1.
6. Use isType(WHILE, j) to check the type of j from the TypeTable to see whether it is of type WHILE or not.
7. If not, continue with the next index from step 1.
8. If yes, ise isType(CALL, i) to check the type of I from the TypeTable to see whether it is of type CALL or not.
9. If not, continue with the next index from step 1.
10. If yes, push i into the vector of answer and continue with the next index from step 1.
11. After iterating through the whole children map, return the vector of answer.
12. If the vector of answer is empty, return the vector with -1 as the only element.

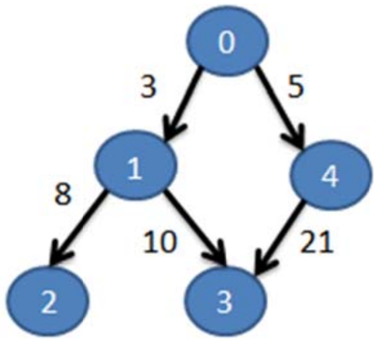
4.3. Design Extractor

The main role of the design extractor is to extract relationships about the SIMPLE program that could not be extracted in the one-time parsing done by the Code Parser. This includes:

a. Extracting information about Modifies and Uses for procedures and for program lines that are calls statements
b. Building the Control Flow Graph (CFG) from the AST and subsequently storing it in the PKB and storing the Next relationship in the PKB

4.3.1. Extracting Relationships

The following shows the steps required to extract relationships like Modifies and Uses for multiple procedures in a single SIMPLE program:

1. Obtain the Calls Table from the PKB. An example of the calls Table is shown below where procedure with index 0 calls procedure with index 1 from program line 3 and procedure with index 4 calls procedure with index 3 from program line 21.
   0 → (1, 3) (4, 5)
   1 → (2, 8) (3, 10)
   2 →
   3 →
   4 → (3, 21)
   This translates into a Calls Tree as follows where the nodes represent procedure indices and the edges represent the program line at which there is a calls statement.
   
   $$\begin{tikzcd} & & 0 \arrow[dl, "3"'] \arrow[dr, "5"] & \\ & 1 \arrow[dl, "8"'] \arrow[dr, "10"] & & 4 \arrow[dl, "21"] \\ 2 & & 3 & \end{tikzcd}$$
2. Run Depth First Search (DFS) on the Calls table (Note: There can be cycles in the graph) in order to obtain a topological sort order of the procedure indices in a queue (below) which starts on the left. Each Queue contains a procedure index and a vector of program lines in which the procedure is called either directly or indirectly.

   (2, [3, 8])   (3, [3, 10])   (1, [3])   (3, [5, 21])   (4, [5])   (0, [ ])

3. Starting from the head of the queue, find all the variables that are modified and all the variables that are used in the procedure. For each of the program lines in the Queue Item, set these program lines (p) to modify and use the respective variables.

For each of the program lines (p), if they are contained in another container statement (c), then set these program lines (c) to modify and use the respective variables too.

4.3.2. BUILDING CFG

Given the AST, the following shows the pseudo code to build the CFG. Then, the CFG is traversed and the Next relationships are stored in the PKB's Next table.

Here are the steps taken to build a CFG:

1. Maintain a currCFGNode pointer.
2. Create a CFG Root.
3. Iteratively traverse each of the procedure type nodes in AST.
    a. For each node, update the currASTNode and currCFGNode. Then create a CFG for stmtLst using the currASTNode.

Here are the steps taken to create a CFG for stmtLst:

1. Iteratively, traverse the children AST nodes and do the following:
    a. If type is ASSIGN, create CFG for Assign.
    b. If type is CALL, create CFG for Call.
    c. If type is WHILE, create CFG for While.
    d. If type is IF, create CFG for If.

To create CFG for Assign or Call, just create a new node and attach to existing CFG.

To create CFG for While, the steps are:

1. Create a new node and attach to existing CFG.
2. Save a pointer to the currCFGNode. This is the toNode for the backwards pointer of the while loop in the CFG that is to be set later.
3. Create the CFG for stmtLst.
4. Find the fromNode for the backwards pointer by running DFS on the currently created CFG.
5. Create a link from the fromNode to the toNode in the CFG.
6. Update currCFGNode.

To create CFG for If, the steps are:

1. Create a new node and attach to existing CFG.
2. For the then:stmtLst, create CFG for stmtLst.
3. Store a pointer to the currCFGNode. This is a leaf node that is to be connected to a dummy -1 node later.
4. For the else:stmtLst, create a CFG for stmtLst.
5. Store a pointer to the currCFGNode. This is another leaf node.
6. Create a dummy -1 node that is connected to both these leaf nodes.
7. Update currCFGNode.

After building the CFG, traverse the CFG and set the Next relationship in the PKB's Next table using the appropriate method. This would allow the Query Evaluator to easily access the Next information without traversing the CFG.

4.4. Query Processor

Query processor consists of three parts: query processor (controller), query parser, and query evaluator.

4.4.1. Query Processor

Query Processor is a façade class for the whole component. The following shows the steps it takes:

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.

Result Projection

Once the results for each relation are computed by the Query Evaluator, the Query Processor does the following steps to project the correct result:

1. Create a table containing tuples of synonyms used during the Query Evaluation.
   To create this table, we iterate through each relation and do one of the following
   a. If none of the synonyms in the relations are already in the table, we set the new table to be the Cartesian product of the current table and the new synonyms
   b. If one or both of the synonyms in the relation are already in the table, we set the new table to be the Join of the current table and the new synonyms
   c. If there are no synonyms in the relation(the relation evaluates to a Boolean value), we do nothing to the table is its true, and clear the table if its false.
2. Augment the table to have synonyms selected by the query but not used in any relation. This is done by setting the new table to be the Cartesian product of the current table and the unused synonyms.
3. Create a set of strings to store the final answers. We use a set to prevent duplicate answer from appearing.
4. For each tuple in the table, create an answer based on the selected synonyms of the query.
5. Return all answers

4.4.2. Query Parser

Query parser has two major functionalities: query validation and query parsing, and they are implemented as functions in the **QueryParser** class. The controller calls query validator to check if the given query is syntactically correct. If it is, query controller will then parse the query by calling the query parser.

Query Validation (Grammar check)

Query validation is done using regular expression method using the grammar rules written in the handbook.

We have considered the option to validate while parsing the query (tokens), but decided to use regular expression to validate the query first then parse afterwards, because of the following reasons:

1. The code is much neater and simpler
2. It is simpler to define the book grammar rule in regex, hence less prone to mistakes
3. It is faster to detect errors rather than parsing and validating (especially if the error is towards the end of the query)
4. Query parsing becomes simpler if we already know the exact possible format of the string query that needs to be parsed.

All types of queries have been defined in a static string, following the grammar rules in the textbook. The strings are then used to validate the queries using regular expression, where the definitions earlier are used. It is very convenient because the grammar rules format in the book is very similar to a regex grammar rules.

As an example, consider this valid query

assign a; while w; Select a such that Follows(w, a) pattern a ("x", _"x+y"_)

Query validator will first break down the query into statements, separated by semicolon. The query above will be broken down into three statements:

1. assign a
2. while w
3. Select a such that Follows(w, a) pattern a ("x", _"x+y"_)

The validator will then use regular expression to check the validity of the statement and retrieve the tokens. Valid declaration statements (design-entity) will be stored into a map with the synonym as the key and its type as the value. This map is called the synonym map and will be used later by the parsing function. In this example, the synonym-map will look as follows:

Synonym	Type
BOOLEAN	BOOLEAN

a	ASSIGN
w	WHILE

This map enables easy look up when the query evaluator evaluates the query. Note that the BOOLEAN type always exists in the synonym table because user can use “BOOLEAN” in his select statement without any declaration.

Select statement will be broken down into such-that, with, or pattern clauses, whose parameters will be checked against the grammar rule.

Consider statement number 3 from the example above. Query validator will use regular expression to check the validity of the statement and retrieve the appropriate tokens.

If the regular expression matching fails, (for example, the number of arguments in the clause is not exactly two) the validator will instantly terminate and declare the query invalid. In the case where statement is valid, all the tokens from the select statement will be stored in a vector, *selectStatement*. This vector will be accessed by the parsing function later on. For efficiency, the *selectStatement* vector will only contain relevant tokens from the statement. Therefore, the unnecessary syntactic punctuation will be removed.

From the example above, the value of *selectStatement[] (dynamic array, c++ vector)* will be:

Select	a	such	that	Follows	w	a	pattern	a	x	_”x+y”_

Query Parsing (Semantic check)

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. constant 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.

With-clause

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.

However, for special cases when the synonym is of type call. There will be another attribute to represent whether call refers to the procedure name (c.procName) or statement number (stmt#). **QueryEvaluator** will then check the additional attribute before evaluating the clause.

Multiple Clauses

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**.

4.4.3 Query Evaluator

General Query Evaluation

Our general query evaluation is as follows:

1. Given 2 synonyms s1, s2 and relation(s1,s2), check if s1 and s2 are already evaluated
2. If they are already evaluated, fetch them from storage and check if relation(s1,s2) is true
3. If they are not fetch them from pkb of the respective synonym type and check if relation(s1,s2) is true
4. Returns all pairs of (s1,s2) such that relation(s1,s2) is true

Query Optimization

To optimize and facilitate a faster running time for each query, we decided to order the relationships in our query object. Below is a summary of which relationships we will evaluate first and last.

Relationships in order of priority

1. With relationships with only 1 synonym	With c.value = 3
2. Pattern relationships with no wildcards	Pattern a(“x”, _”x+y”_)
3. Relationships with no synonyms	Modifies(5, “x”), Follows(1, 4), Parent(_,4)
4. Direct Relationships with only 1 synonym	Modifies(a, “x”), Follows(s, 4), Parent(s, _)
5. Indirect Relationships with only 1 synonym	Follows*(s1,5), Parent*(s1,8)
6. Direct Relationships with 2 synonyms	Uses(s, v), Next(n, n2)
7. Indirect Relationships with 2 synonyms	Next*(n1, n2)
8. Affects* Relationships with 2 synonyms	Affects*(a1,a2)

Query Clause Evaluation Examples

```
stmt s1; Select s1 such that Follows(s1, 4)
```

As s1 first appears in Follows(s1, 4), QE will fetch all valid statement numbers by calling PKB method getAllStmts(Type Stmt) and check that for each value of s1 does Follows(s1,4) evaluate to true.

```
variable v; Select v such that Modifies(5,v) with v.varName = "x"
```

Here, we have 2 relationship clauses: Modifies and with. As with has a higher priority, we move it to the top and evaluate it first. We then store “x” as the evaluated answer for v. Hence when we move on to evaluate Modifies(5,v), we do not pull all valid variables but only the previously evaluated “x”.

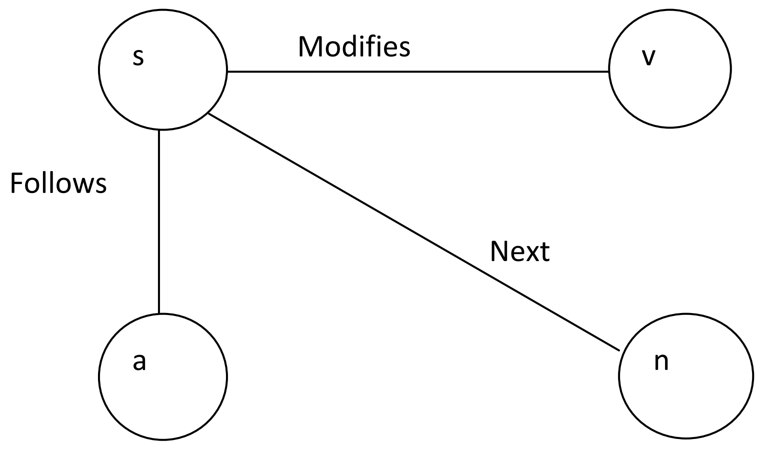
Alternative Query Data Representation

Previously in our earlier reports we mentioned that we utilized a table to store the information each relationship has with each synonyms. For instance, Modifies(s, “v”) contains the synonym s, and hence we will store in our table “s is linked to Modifies(s,”v”)”.

An alternative representation of this will be to draw a graph and consider all synonyms as vertices and relationships as edges. To illustrate, consider the following example

Select BOOLEAN such that Modifies(s,v) and Follows(s,a) and Next(s, n)

The output graph would be:



Affects and Affects*

Given the two arguments(two assignment statement numbers a1 and a2), Affects(a1, a2) is evaluated as follows:

1. We check the memo-table if Affects(a1, a2) has been calculated. If it has, return the stored answer, else continue to step 2.
2. Check whether a2 uses the variable modified by a1. This check is done first as it can be done in O(1) and will eliminate many trivial cases thus saving precious time.
3. We perform a modified-DFS on the CFG starting at a1. In this modified DFS, we have an extra condition of expanding a node only if it does not modify the variable modified in a1. We terminate if we reach a2. This runs in O(|s|), where s is the number of statements in the procedure.
4. If our DFS reaches a2, we return true for Affects(a1, a2). Else, we return false. For either result, we store the result in a memo table.

By storing each result in a memo-table(which is maintained for the duration of the query evaluation), we stop any redundant calculations and return the answers in O(1).

Affects* is evaluated in a similar way as Affects. Given assignment statements a1 and a2, we evaluate Affects*(a1, a2) as follows:

1. We check the memo-table if Affects*(a1, a2) has been calculated. If it has, return the stored answer, else continue to step 2.
2. We perform a modified-DFS on the CFG starting at pair(-, a1). In this modified DFS, we have two change:
   a. When expanding a node, we add the pairs (currNode, n) for all n where Next*(currNode, n) is true.
   b. We only expand a node if it's the initial node(pair(-, a1)) or if given pair(x, y), Affects(x, y) is true.

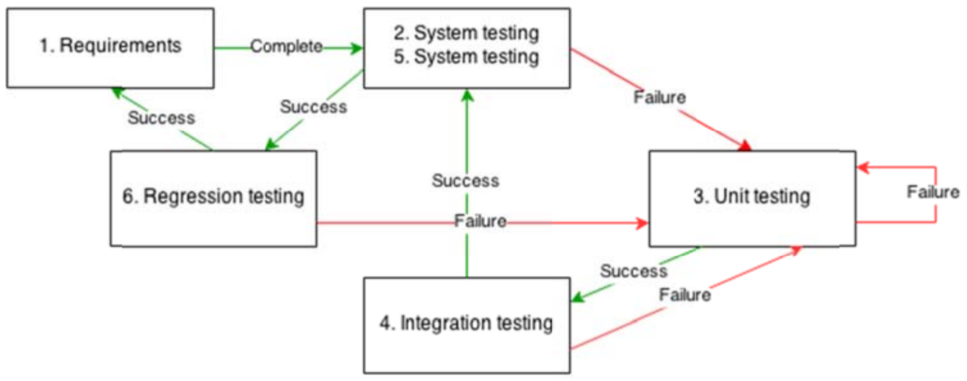
   We terminate if we reach a pair(x, y) where y==a2.
3. If our DFS reaches a2, we return true for Affects*(a1, a2). Else, we return false. For either result, we store the result in a memo table.

Once again, a memo-table is kept to prevent redundant calculations.

5. Testing

5.1. Overview

Throughout 4 iterations, the team followed the following testing cycle:



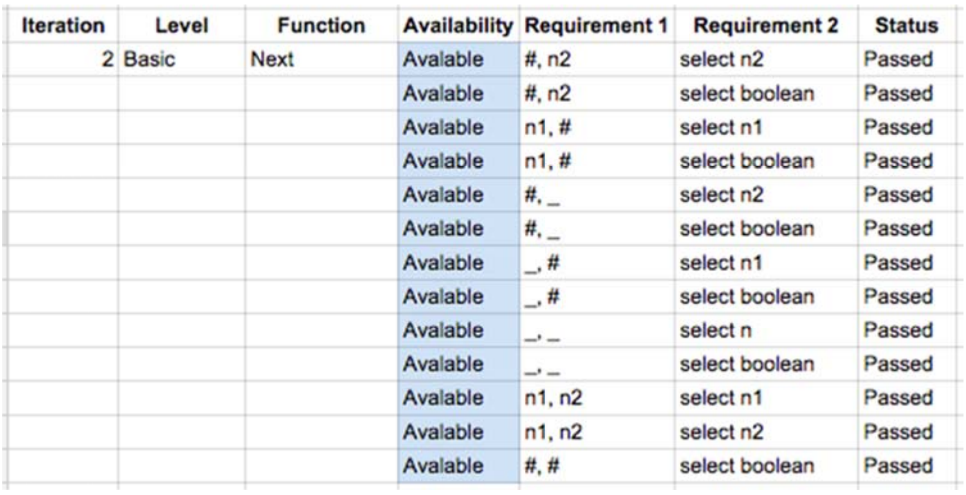
The cycle is started with the creation of system testing scripts from the current iteration's requirements by the test lead. These test scripts will become executable requirements, which will be referenced by other team members. As the development progresses, each team member is responsible for his/her own unit testing. As soon as two or more components pass the unit testing, integration testing will be carried out. The success of this integration testing will prompt another run of system testing. Upon successful completion of system testing, the testing cycle is concluded by regression testing. This is done by running all of the past and current test cases to make sure that the new feature does not interfere with other features.

5.2. Testing Standards

To ensure thorough test coverage, the following testing standards are observed for each test suite.

5.2.1. Feature-centric testing

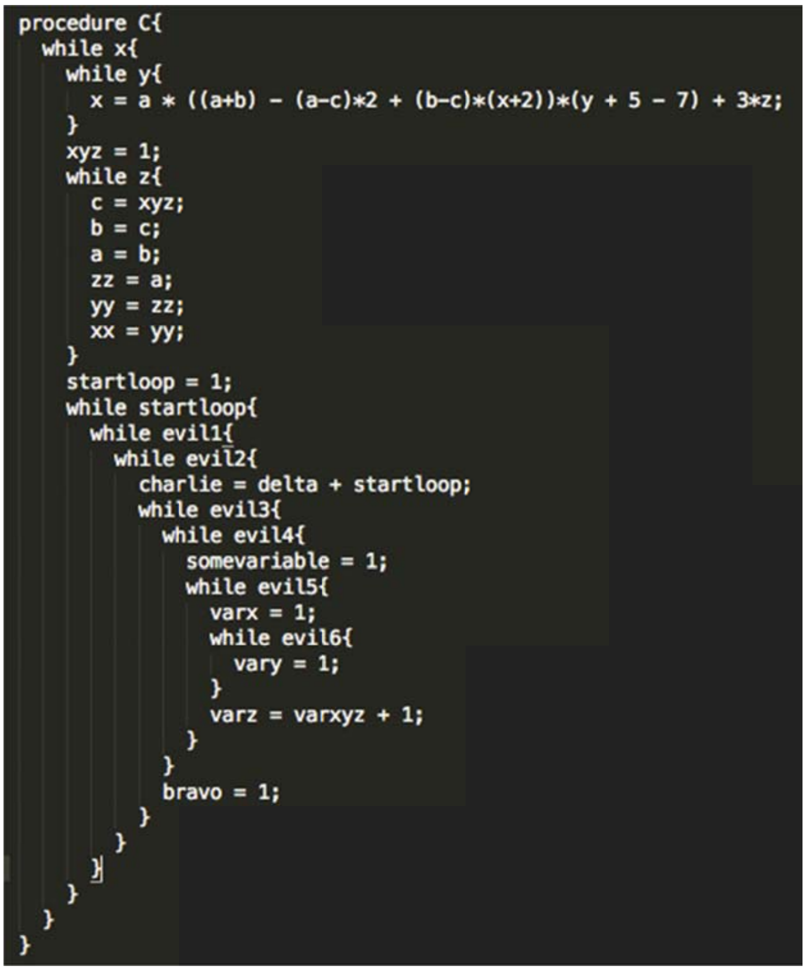
Each test suite needs to have feature-centric testing that tested all the possible combinations of valid query parameters and interactions between relationships. The first layer of the testing is to ensure the correctness of the program. The table below illustrated the various combinations of parameters needed to verify the correctness of Next relationship. This thorough coverage of test cases ensures that the deployed feature is complete.



Iteration	Level	Function	Availability	Requirement 1	Requirement 2	Status
2	Basic	Next	Avalable	#, n2	select n2	Passed
			Avalable	#, n2	select boolean	Passed
			Avalable	n1, #	select n1	Passed
			Avalable	n1, #	select boolean	Passed
			Avalable	#, _	select n2	Passed
			Avalable	#, _	select boolean	Passed
			Avalable	_, #	select n1	Passed
			Avalable	_, #	select boolean	Passed
			Avalable	_, _	select n	Passed
			Avalable	_, _	select boolean	Passed
			Avalable	n1, n2	select n1	Passed
			Avalable	n1, n2	select n2	Passed
			Avalable	#, #	select boolean	Passed

5.2.2. Stress testing

The second layer of testing is the boundary testing. In this case, the team used upper boundary testing to ensure system reliability under high-stress query execution. It is important for the team to make sure that the system does not crash under tough query scenarios. The excerpt of source code below is one example of what the team used to stress-test the system. It involves many procedures in one source code, and each procedure has complicated operations and nesting levels.



```
procedure C{
  while x{
    while y{
      x = a * ((a+b) - (a-c)*2 + (b-c)*(x+2))*(y + 5 - 7) + 3*z;
    }
    xyz = 1;
    while z{
      c = xyz;
      b = c;
      a = b;
      zz = a;
      yy = zz;
      xx = yy;
    }
    startloop = 1;
    while startloop{
      while evil1{
        while evil2{
          charlie = delta + startloop;
          while evil3{
            while evil4{
              somevariable = 1;
              while evil5{
                varx = 1;
                while evil6{
                  vary = 1;
                }
                varz = varxyz + 1;
              }
            }
            bravo = 1;
          }
        }
      }
    }
  }
}
```

This source code is accompanied by queries similar to the following, that tests the consistency of system's behavior when the projected result set is huge, and/or the computation is complicated.

```
assign a1,a2,a3,a4,a5,a6,a7,a8;
Select <a1,a2,a3,a4,a5,a6,a7,a8> such that Follows*(a1,a2) and
Follows*(a3,a4) and Follows*(a5,a6) and Follows*(a7,a8)
```

5.2.3. Optimization testing

The third layer of testing is the optimization testing. This test becomes vital after the introduction of multiple clause queries. The aim of this test is to ensure that Query Evaluator is intelligent enough to reorder clauses in a query so that the performance of the system is comparable to that of the optimal evaluation strategy.

The test cases are chosen in such a way that there is one optimal order in which Query Evaluator can answer the query. One example is the following query.

```
procedure p1, p2; stmt s1, s2; prog_line n1, n2; variable v;
Select BOOLEAN such that Calls* (p1, p2) and Next* (n1, n2) and Uses (s1,
v) and Affects*(1, 30)
```

This query is filled with complicated relations such as Next* and Affects*. However, one of these clauses will return false, which in turn will default the result of the query to be "false". The following query is another example to illustrate selective evaluation of clauses in a query. The optimal strategy would be to evaluate "with s7.stmt# = 29" first before other clauses.

```
procedure p1, p2; stmt s1, s2, s3, s4, s5, s6, s7;
Select s7 such that Calls*(p1, p2) and Modifies (s1, v1) and Uses (s2, v2)
and Modifies (s2, v2) and Affects* (s2, s3) and Affects* (s3, s4) and
Parent* (s5, s6) and Modifies (s7, v) with s7.stmt# = 29
```

5.2.4. Positive, negative, and disruptive testing

Each test suite is required to have a combination of positive, negative, and disruptive cases. Following this token, the test suite needs to expose the system to queries that will return:

- Positive queries: one answer, some answers, and a lot of answers
- Negative queries: no answer or "false"
- Disruptive queries: no answer due to invalid syntax or query logic

5.2.5. Separation of concerns

For a better project management, test cases are organized into independent modules that each serves one testing purpose. This is to help the team to focus on one sub-feature in each test file. This in turn allows the team to identify team member who is in charge of test case failure. This separation of concerns allowed the team to identify point of failure, quickly fix bugs, and efficiently re-test affected subsystem.

The test files are organized based on difficulty:

- Basic: typically one feature (clause) in a query, to test the correctness of each feature
- Intermediate: a combination of two features (clauses) in a query
- Advanced: any number of features (clauses) in a query with complicated logic and operations

Invalid test cases are also housed in their own test file, independent of other features. This is to give confidence to the team that errors in other test files are not caused by mishandling of invalid queries.

All in all, with all of the aforementioned testing standards, the team has maintained a thorough and comprehensive testing environment.

6. Coding Standards

Our entire team adopted similar coding standards to maintain a common coding quality. Some of the coding standards are listed below:

1. Indentation and whitespace
    a. Use it to indicate different code segments
2. Comments to enhance understanding and communication
3. Descriptive variable declarations
    a. Always start with lower case
    b. Use CamelCase
    c. Use only letters and numbers
4. Informative function naming conventions
    a. All getters start with “get”
    b. All setters start with “set”
    c. All functions that start with “is” returns a Boolean value
5. Keep it simple and effective
    a. Avoid complex code fragments
6. Refactoring

Standards between abstract APIs and concrete APIs

The correspondence between the abstract APIs and the concrete APIs was improved by:

1. Ensuring that the abstract APIs provide an interface for the concrete APIs to be built upon.
2. Making abstract APIs as comprehensive as possible by:
    a. Offering an extensive description and responsibility of the abstract API
    b. Specifying the complete parameters needed for the function

7. API

Please view our Doxygen at:

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