

# **CS3202: Software Engineering Project II**

## **Iteration and Assignment 3**

9th Mar 2015

**Team Number: 10**

**Consultation Day/Hour: Monday, 11.00-11.45am**

### **TEAM MEMBERS**

#### **Parser & Design Extractor:**

Saw Han Qiang Matthew	A0097556M	a0097556@u.nus.edu
-----------------------	-----------	--------------------

#### **Program Knowledge Base:**

Joanne Mah Jia Wen	A0080412W	a0080412@u.nus.edu
--------------------	-----------	--------------------

William Kimsha	A0100234E	a0100234@u.nus.edu
----------------	-----------	--------------------

#### **Query Processor:**

Boo Kuok Liang	A0087547N	a0087547@u.nus.edu
----------------	-----------	--------------------

#### **Testing:**

Chen Fang	A0091613L	a0091613@nus.edu.sg
-----------	-----------	---------------------

# **1.Achievements and problems in this iteration**

## **Scope of SPA implemented**

This iteration has introduced the following new features.

- Affects, Affects\*
- Optimiser for PKB

The support for tuples within Query Processor has been implemented in iteration 2. Within this iteration, we came up with a test plan to provide better testing coverage for it. Other than that, we have also improved some algorithms for the intermediate result table and changed some of the data structure to make it more efficient.

We have also successfully fixed the bugs for the bit vector operations used in PKB in this iteration.

## **Special achievements and/or problems**

On top of getting Affects to work, we have explored various methods to optimise Affects. These methods will be elaborated in section 4. Documentation of important design decisions.

## 2. Project Plans

### 2.1 Plan for the whole project

	Iteration 1 (2 weeks)			Iteration 2 (4 weeks)				
	Full parser	Call stmts	Patch up BQE	PQL With clause	Design extractor	PKB Next	Multiple clauses	PKB counter
Matthew	*				*			
Joanne		*				*		*
Kuok Liang			*	*			*	*
William	*					*		
Chen Fang			*				*	

	Iteration 3 (3 weeks)				Iteration 4
	PQL Affects	PKB Affects	Tuple results	Optimisation	Bonus Feature
Matthew		*			*
Joanne		*			
Kuok Liang	*		*	*	*
William		*			*
Chen Fang	*		*	*	
Zaki		*		*	*

## 2.2 Plan for this development iteration

### Week 1 (10 Mar - 16 Mar)

	Iteration 3			
	Integration Testing: DE → PKB	Planning for Affects	BitVector	Improvements for StmtTypeTable and Modifies
Matthew	*	*		
Joanne		*		*
William			*	
Kuok Liang		*		
Chen Fang				

### Week 2 (16 Mar - 23 Mar)

	Iteration 3			
	Improvement for IntermediateResult	Affects	Integration Testing: DE → PKB	Integration Testing: Affects
Matthew			*	
Joanne		*		*
William				
Kuok Liang	*			
Chen Fang	*			

### Week 3 (23 Mar - 30 Mar)

	Iteration 3			
	Optimiser	Affects*	Validation Testing	General Testing
Matthew		*	*	
Joanne			*	
William			*	
Kuok Liang	*		*	
Chen Fang				*

### 3.UML diagrams

#### Implementation of CreateBitVector

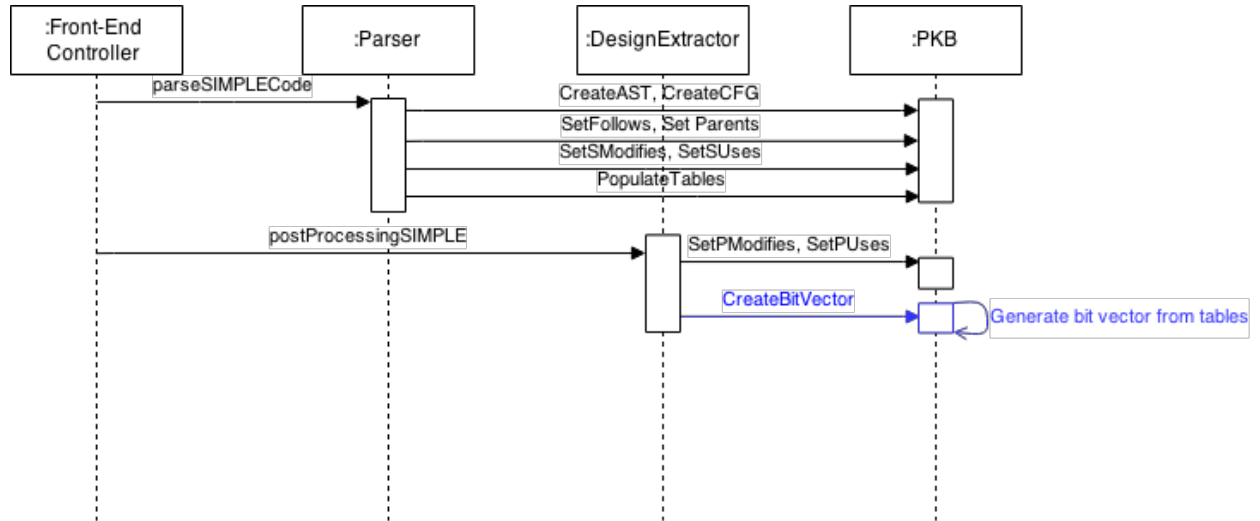


Fig. 1 UML sequence diagram for implementation of `CreateBitVector`

## Flow Chart for Affects

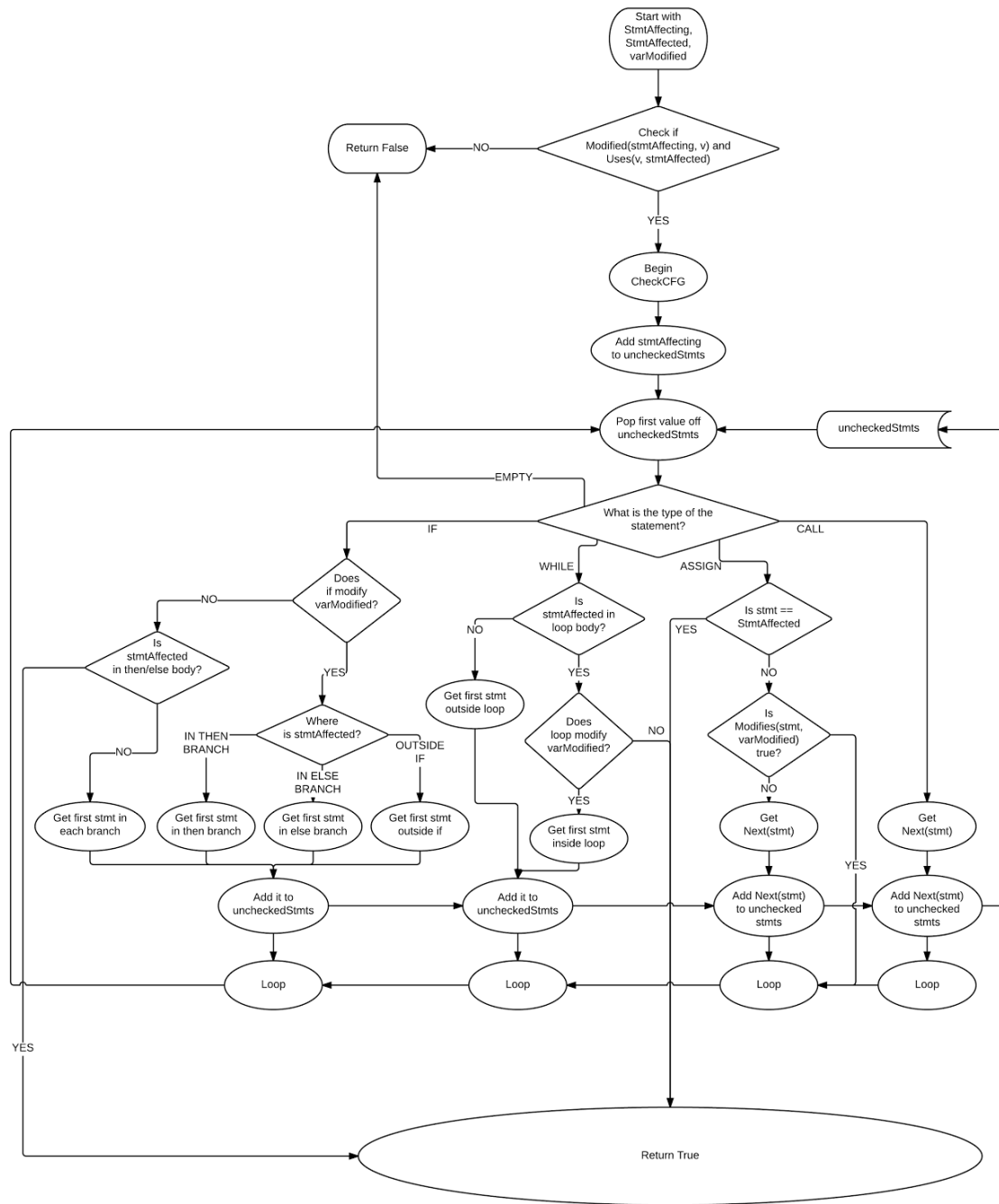


Fig. 2 UML flow diagram for implementation of Affects

## How Affects interacts with other components

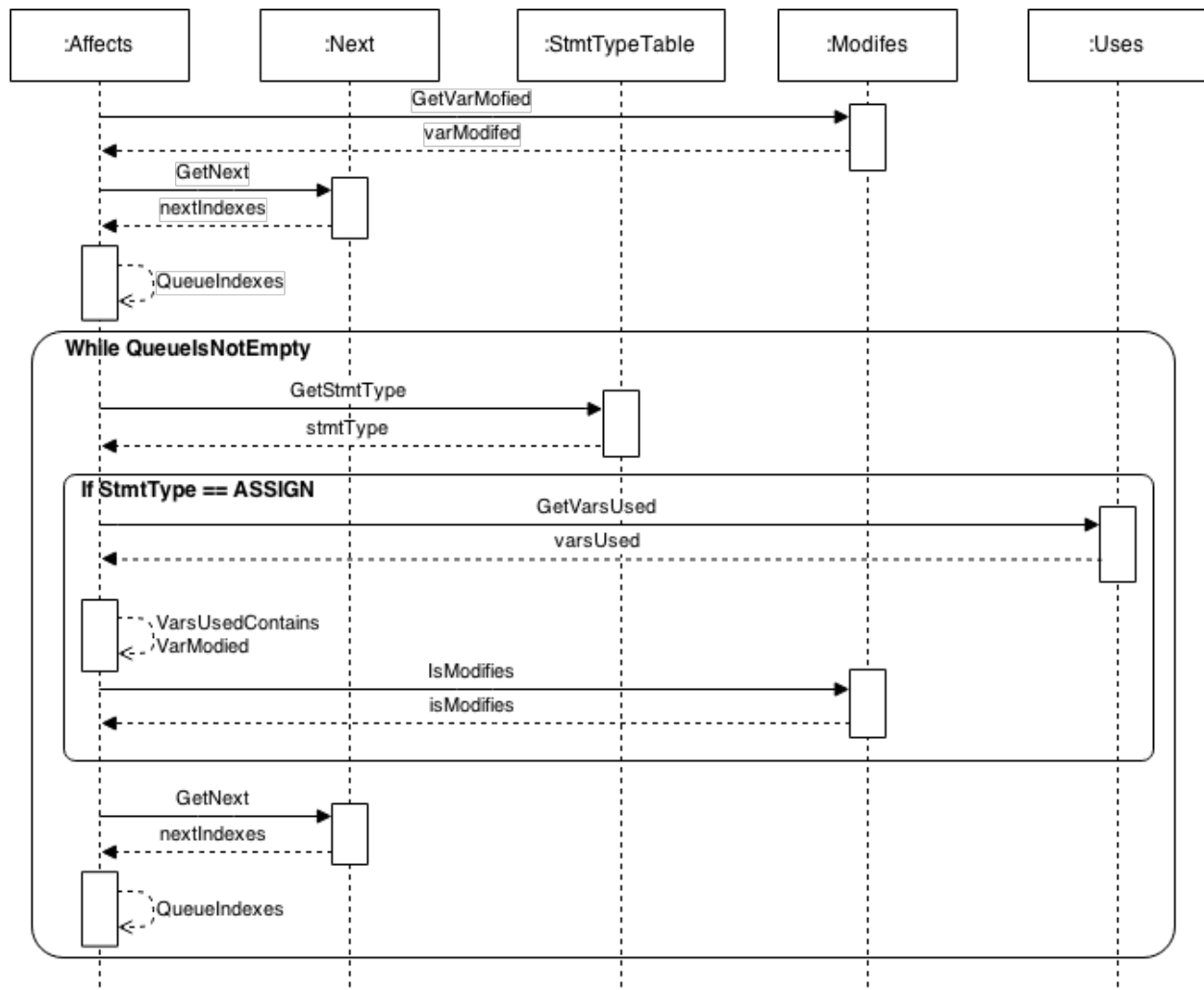


Fig. 3 UML sequence diagram for interactions between Affects and other components

## 4.Documentation of important design decisions

### Affects

#### Basic implementation

As shown in *Fig. 3* in the Section 3 for UML diagrams, we see that Affects is a controller that computes a given relationship by retrieving values from the tables stored in the Modifies and Uses components. It also invokes the Next class in order to traverse the CFG, as well as the StmtTypeTable to check for the type of statement of a given statement number.

Traversing the CFG is carried out each CFG node recursively. For each method call, the variable which have been modified at a1 or the list of variables which has been used at a2, depending on whether you're traversing up or down the CFG, will be passed on for checking.

At each node, the statement type is first checked. If the statement was an assign statement

#### Optimisation strategies

Affects can be optimised through assertions as well as minimising traversal of the CFG. The former would be to verify if the basic conditions for a1 and a2 were met, without having to traverse the CFG. This are checks that:

- Statements a1 and a2 are assign-type statements
- a1 modifies a variable v that is being used by a2

During tree traversal, the branching of the source code's control flow would mean that several nodes get revisited via different paths. Along these paths, different variables may get modified in the process. The abstract of how our program can will bypass these alternative paths will be elaborated in the following sections.

#### While statements

If a while statement does not given variable(s)<sup>1</sup>, <V>, none of the statements contained within the while container statement modifies any of <V>.

Therefore, during a downwards traversal of the CFG, we could bypass the entire statement by going to the statement following the while statement if a2 does not reside within the while body. On the other hand, if the statement does modify any of the variables(s) in <V>, we will invoke `GetStmtsInContStmt ModifyingVar(int contStmt, int varModified)` which will allow us to jump to the first statement modifying <V>.

On the other hand, an upwards traversal would require a different implementation of the above method. Since there may be the change the thread enters the while loop first, we need to check first, for each statement, if the statement following before it is a while statement. If it is, we invoke the same procedure as above but in reversed direction.

---

<sup>1</sup> For an downwards traversal of the CFG, we would only need to check if a single variable modified in a1 is modified again in the path between a1 and a2. However, for an upwards traversal, we would have to check if a list of variables used in a2 has been modified between a1 and a2.



### If-then statements

Similar to the while statement, traversal through the body of an if-then statement can be bypassed if the statement does not modify any of the variables in consideration. However, if the statement does modify any of the variables in consideration, we would have to check if the variable(s) belong in the “then-body” or the “else-body”, or both. We would also have to check if `a2` appears within any of these segments, and if so, whether it appears before or after the statements modifying `<V>`.

### Affects\*

The difficulty in implementing Affects\* was attempting to find an algorithm that was efficient and correct, and expressing that algorithm in a way that made the most sense. We attempted to use multiple programming paradigms to implement this section. Additionally, we managed to find some useful optimisations, that were not only easy to implement, but provided a notable performance boost (bang for our ‘buck). // tree trimming

### Basic implementation

In the implementation of Affects\*, we went with a simple looping algorithm. The algorithm checks if the statement in question uses the variables modified by an earlier statement. If yes, we can extend the Affects\* set and look for statements that use the variable modified by this new statement. Because of the definition of Affects, we can use a greedy algorithm and assume the first statement encountered that meets the criteria is part of the Affects\* set.

The problem comes when we attempt to follow the Next path through conditionals. While loops present a problem of infinite loops, and if/then/else statements prevent us from using a proper loop, without some means of keeping track where the branches are (which can get out of hand easily in large programs). Therefore, Affects\* was implemented in a tail-recursive manner. Tail-recursive functions can be mapped one-to-one to an equivalent loop, and allow us to express the following of split Next paths using recursion, which is natural and easy to understand.

### Optimisation strategies

Optimisation was done using tree trimming. By checking if while and if statements were using the variable in question, we are able to trim the unreachable branches along these conditionals and save time. This is similar to the optimisation strategy for Affects.

Additionally, the result was memoised. Since a depth-first algorithm was used, if the algorithm encountered a statement that was computed for Affects\* in a previous branch, it would return immediately, since the result would have already been computed (each assignment statement only modifies 1 variable, and will always have the same set of Affects\*). Upon observation, we see that the problem has optimal substructure and recurring subproblems, which is easily optimised by dynamic programming techniques, hence memoisation.

## 5.Coding standards and experiences

Additional API in StmtTypeTable and Modifies to easily retrieve the intermediate results

required for computing Affects.

- `GetStmtTypeOf(int stmtIndex)`
- `GetStmtsInContStmtModifyingVar(int contStmt, int varModified)`

Additional API in PKB to retrieve and interact with bit vectors (arguments are omitted):

- |   |   |
|---|---|
| ● <code>Is[Stmt/Proc]ModifiesBV</code>        | ● <code>Is[Stmt/Proc]UsesBV</code>        |
| ● <code>Get[Stmt/Proc]ModifyingStmtBV</code>  | ● <code>Get[Stmt/Proc]UsingStmtBV</code>  |
| ● <code>Get[Stmt/Proc]ModifiedByStmtBV</code> | ● <code>Get[Stmt/Proc]UsedByStmtBV</code> |

Even though `Is[Stmt/Proc]ModifiesBV` and `Is[Stmt/Proc]UsesBV` perform the same functions as `Is[Stmt/Proc]Modifies` and `Is[Stmt/Proc]Uses`, the formal can only be called after postprocessing of the SIMPLE code has been completed by DE. Thus, the different names will be used to avoid calling the wrong methods.

## 6. Query processing

### 6.1 Validating program queries

- Keywords allowed

Relationships	Synonyms	Miscellaneous	Attribute Name	Signs
Modifies	assign	Select	value	.
Uses	stmt	BOOLEAN	stmt#	=
Parent	while	and	varName	—
Parent*	if	such that	procName	“”
Follows	variable	pattern		()
Follows*	constant	with		,
Calls	procedure			;
Calls*	prog_line			+
Next				-
Next*				*
Affects				
Affects*				

- Check if the structure of the query is valid. Structure must be one of the following, e.g.,  
Declaration-clause Select-Clause  
Declaration-clause Select-Clause (SuchThat-clause\* | Pattern-clause\* | with-clause\*)
- Keywords defined in the table above cannot be declared as synonym.
- For each synonym in the query, check whether the synonym is declared in declaration.
- Checking if Such That-clause, Pattern-clause, and With-clause have valid syntax and correct number and types of arguments, as defined in PQL definition in Handbook. These clauses are checked against their corresponding table as we're using table driven approach.
- Only signs defined in the table above can be used in a query.
- Character sequences other than keywords are treated as synonym and will be checked for validity (whether it has been declared, and if it is a valid argument).
- Character sequences enclosed in “” are treated as identity or expression

- Character sequences enclosed in \_“”\_ are treated as expression

New validation Rule for this iteration:

- New keyword allowed - Next, Next\*, and, if
- Pattern clause includes if and while
- As we're using table driven approach to query validation, we have extended the relationship table to include Next and Next\*, and entity table to include if, and pattern table to include if and while
- Multiple clauses are allowed, check if the clauses between “and” are of the same type, e.g.,

Modifies(..) and Uses(..) - Valid

pattern a(..) and pattern if(..) - Valid

Modifies(..) and pattern a(..) - Invalid

Modifies(..) and a1=2 - Valid

1	assign
2	stmt
3	while
4	if
5	variable
6	constant
7	prog_line
8	procedure

**Entity Table**

Relationship	No. of Arguments	Type of Argument 1	Type of Argument 2
Modifies/ Uses	2	<ul style="list-style-type: none"> <li>• synonym (procedure   assign   stmt   while   prog_line)</li> <li>• string</li> <li>• integer</li> </ul>	<ul style="list-style-type: none"> <li>• synonym (variable)</li> <li>• string</li> <li>• –</li> </ul>
Parent/ Parent*	2	<ul style="list-style-type: none"> <li>• synonym (stmt   while   prog_line)</li> <li>• integer</li> <li>• –</li> </ul>	<ul style="list-style-type: none"> <li>• synonym (assign   stmt   while   prog_line)</li> <li>• integer</li> <li>• –</li> </ul>
Follows/ Follows*	2	<ul style="list-style-type: none"> <li>• synonym (assign   stmt   while   prog_line)</li> <li>• integer</li> <li>• –</li> </ul>	<ul style="list-style-type: none"> <li>• synonym (assign   stmt   while   prog_line)</li> <li>• integer</li> </ul>
Calls/Calls*	2	<ul style="list-style-type: none"> <li>• synonym (procedure   <del>prog_line</del>)</li> <li>• string</li> <li>• <del>integer</del></li> </ul>	<ul style="list-style-type: none"> <li>• synonym (<del>procedure</del>   prog_line)</li> <li>• string</li> <li>• <del>integer</del></li> </ul>
Next/Next*	2	<ul style="list-style-type: none"> <li>• synonym (prog_line)</li> <li>• integer</li> <li>• –</li> </ul>	<ul style="list-style-type: none"> <li>• synonym (prog_line)</li> <li>• integer</li> <li>• –</li> </ul>
Affects/ Affects*	2	<ul style="list-style-type: none"> <li>• synonym (assign   <del>prog_line</del>)</li> <li>• integer</li> <li>• –</li> </ul>	<ul style="list-style-type: none"> <li>• synonym (assign   <del>prog_line</del>)</li> <li>• integer</li> <li>• –</li> </ul>

Relationship Table

Pattern Synonym	No. of Arguments	Argument 1	Argument 2	Argument 3
assign	2	<ul style="list-style-type: none"> <li>• synonym (variable)</li> <li>• string</li> <li>• _</li> </ul>	<ul style="list-style-type: none"> <li>• expression</li> </ul>	
while	3	<ul style="list-style-type: none"> <li>• synonym (variable)</li> <li>• string</li> <li>• _</li> </ul>	<ul style="list-style-type: none"> <li>• _</li> </ul>	<ul style="list-style-type: none"> <li>• _</li> </ul>
if	3	<ul style="list-style-type: none"> <li>• synonym (variable)</li> <li>• string</li> <li>• _</li> </ul>	<ul style="list-style-type: none"> <li>• _</li> </ul>	<ul style="list-style-type: none"> <li>• _</li> </ul>

Pattern Table

No. of Arguments	Type of Argument 1	Type of Argument 2
2	<ul style="list-style-type: none"> <li>• attrRef (procName)</li> </ul>	<ul style="list-style-type: none"> <li>• attrRef (procName)</li> <li>• string</li> </ul>
2	<ul style="list-style-type: none"> <li>• attrRef (varName)</li> </ul>	<ul style="list-style-type: none"> <li>• attrRef (varName)</li> <li>• string</li> </ul>
2	<ul style="list-style-type: none"> <li>• attrRef (stmt#)</li> <li>• synonym (prog_line)</li> </ul>	<ul style="list-style-type: none"> <li>• attrRef (stmt#   value)</li> <li>• synonym (prog_line)</li> <li>• integer</li> </ul>

With Table

## 6.2 Design and implementation of query evaluator

### Data Representation

After parsing and validating the query, a query tree is constructed. Our implementation of the query tree is a class (QueryData) that consists of 5 vectors, each vector stores the information of each clause, namely,

- declarations
- selectClauses
- suchThatClauses
- patternClauses
- withClauses

When evaluating query, the BQE will iterate through each of these vectors so evaluating multiple clauses is possible.

In each vector, the clauses are made of struct that contains the information of that particular clauses, such as type and value of the 2 arguments, and the relationship in the case of SuchThatClause. An example struct that define the SuchThatClause is shown below:-

```
struct SuchThatClause {
    RelationshipType relationship;
    Argument arg1, arg2;
};
```

The RelationshipType type is an enum that contains all valid relationship:-

```
enum RelationshipType {
    MODIFIES,
    USES,
    PARENT,
    PARENTT,
    FOLLOWS,
    FOLLOWST,
    CALLS,
    CALLST,
    NEXT,
    NEXTT,
    AFFECTS,
    AFFECTST,
    INVALID_RELATIONSHIP_TYPE
};
```

where the Argument type is another struct specifying the argument type (SYNONYM/INTEGER/UNDERSCORE/etc.) and the argument value.

To define the grammar of PQL, we used enum to store keywords like the RelationshipType above, other than that, we also have enum for ArgumentType, SynonymType. This, combine with the struct that define the structure of each clause, is defined in Grammar.h. As the name implied, it is just a header file that store the PQL grammar, this way of separating grammar and validation/evaluation allow us to make changes to grammar without affecting the BQE. For example, adding Next and Affects in query in the future can be done by simply uncommenting them in the RelationshipType enum above.

## Strategy for Basic Query Evaluation (BQE)

The BQE is not optimized, so clauses are evaluated from left to right.

We've decided to change our data structure for intermediate result from table to lists that have pointer to each other. Each synonym has its own list (analogous to a column in a table), and relationship between list is achieved by using pointer. Element from different lists point to each other to form a combination of valid result (analogous to a row in a table). This way, double computation is eliminated when evaluating query with clauses depended on each other like Select a such that Modifies(a,v) and Parent(w,a). Besides, this made selecting tuple result possible, e.g. select <a,w,v> , as each row correspond to a valid tuple result.

The intermediate result table is a private data, and all evaluating function that requires it can access it.

Consider this short SIMPLE program and the following query, we will go through the step of evaluation using list.

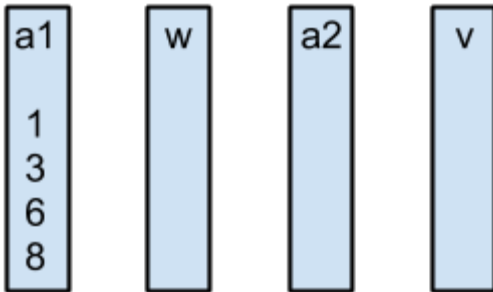
```
1      x = 1
2      while a {
3          x = x + 1
4          y = x + 2
5          while b {
6              x = x + y
              }
          }
7      while c {
8          x = y + 1
          }
```

assign a1,a2;while w;variable v;

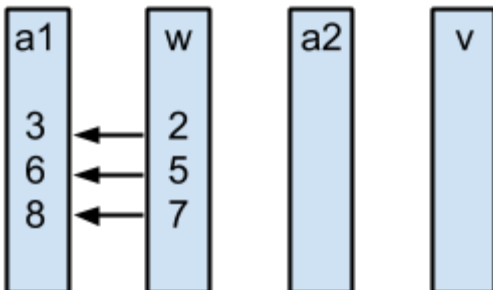
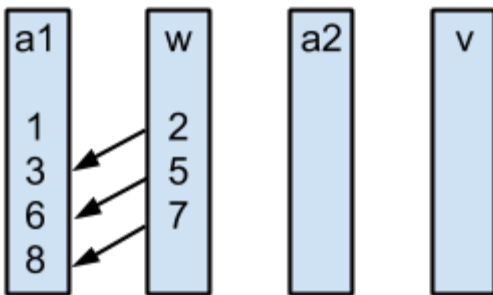
Select a1 such that Modifies(a1,"x") and Parent(w,a1) pattern a2("x","y+1")  
with a1 = a2



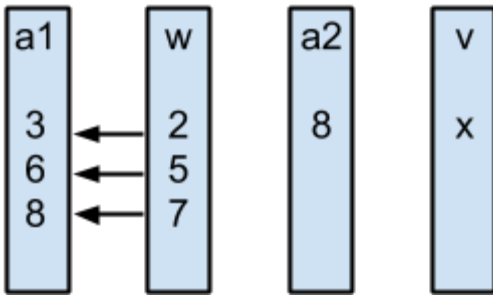
1. The BQE will evaluate  $\text{Modifies}(a1, "x")$  first, since the result for  $a1$  is empty, it will populate it with 1,3,6,8.



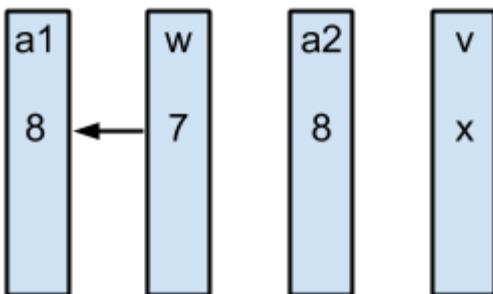
2. Then it proceeds to evaluate  $\text{Parent}(w, a1)$ , find the pairs (2,3), (5,6), (7,8) to be valid. So it points 2->3, 5->6, 7->8, and since nothing points to 1, it will be deleted.



3. Then it proceeds to evaluate pattern  $a2(v, "y+1")$ , find that  $a2 = 8$  and  $v = x$ , so it populates  $a2$  with 8 and  $v$  with  $x$ .



4. Lastly, it evaluates with  $a1 = a2$ , further narrow the list of  $a1$  down to 8, and remove the other invalid pair.



## Optimisation

Optimiser is a separate component within Query Evaluator. Its purpose is to reorder clauses in a query to improve evaluation time for the entire query. Currently, optimisation is done by selecting which clauses can be evaluated first according to the following rules:

- most specific entity types first
- smallest expected intermediate results first, based on number of each relationships stored in the PKB
- shortest expected evaluation time of a given relationship first

The above rules can vary across clauses depending on a mixture of factors:

- information given about the entities or synonyms in a clause (e.g. "Select a" vs "Select s")
- nature of the query (e.g. retrieving or computing relationships, pattern matching...)
- the number of entities in the parsed source code (e.g. if there were more call statements than assignment statements, we should evaluate the assignment statements first)

We gave weights for each attribute in a each clause with the heaviest weight given to attributes which were expected to take the longest to compute. The clauses are then sorted according to their weights such that clauses which take the fastest to compute are evaluated first.

## 7. Testing

### Unit Testing

Unit testing for PKB for this iteration focusses on verifying operations involving the bit vectors. Tests are similar to and reused from their non-bit vector counterparts.

#### PKB: Test 1

```
Modifies::SetStmtModifiesVar(1, 0);
Modifies::SetStmtModifiesVar(2, 0);
Modifies::CreateBitVector();

CPPUNIT_ASSERT(Modifies::IsStmtModifyingVarBV(1,0));
CPPUNIT_ASSERT(Modifies::IsStmtModifyingVarBV(2,0));
```

**Test Purpose:** Proper insertion of Modifies relationship into the Modifies BitVector.

**Required Test Inputs:** Statement and variable indexes.

**Expected Test Result:** Correct assertion statements.

#### PKB: Test 2

```
Calls::SetCalls(1, 0);
Calls::SetCalls(2, 0);
Calls::CreateBitVector();

CPPUNIT_ASSERT(Calls::IsCallsBV(1,0));
CPPUNIT_ASSERT(Calls::IsCallsBV(2,0));
```

**Test Purpose:** Proper insertion of Calls relationship into the Modifies BitVector.

**Required Test Inputs:** Statement and procedure indexes.

**Expected Test Result:** Correct assertion statements.

Unit testing for PQL will focus on the basic features of tuples. More complicated tests will be carried out in integrated and validation testing where other components of SPA are involved.

#### PQL: Test 1

```
IntermediateResult::InsertPair(a , 4 , w , 3)
IntermediateResult::InsertPair(a , 5 , w , 3)
IntermediateResult::InsertPair(a , 6 , w , 3)
IntermediateResult::InsertPair(a , 4 , v , "x")
IntermediateResult::InsertPair(a , 5 , v , "x")
IntermediateResult::InsertPair(a , 6 , v , "y")

std::string query = "assign a;while w;variable v;Select <a,w,v>"
QueryData queryTree;
queryTree.ProcessQuery(query)
```

```

std::vector<std::string> expectedResult;
std::list<std::string> actualResult;
expectedResult.push_back("4 3 x");
expectedResult.push_back("5 3 x");
expectedResult.push_back("6 3 y");

QueryEvaluator::EvaluateQuery(queryTree, expectedResult);
CPPUNIT_ASSERT(QueryEvaluator::IsEqual(expectedResult, actualResult));

```

**Test Purpose:** To test whether Select tuple returns the correct result.

**Required Test Inputs:** Pairs of linked result to form the intermediate result table.

**Expected Test Result:** Correct tuple result.


## Integration Testing

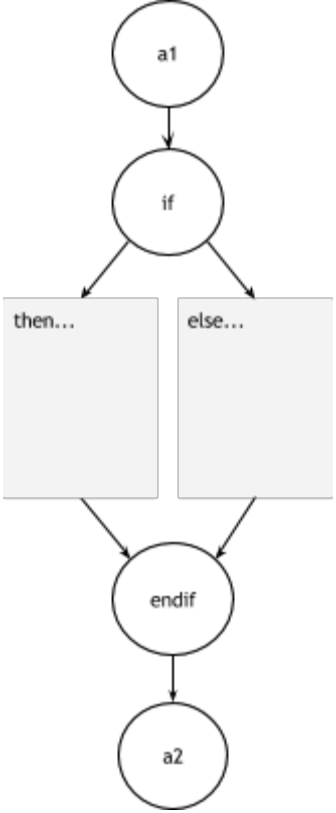

### Test plan for Affects

Given Affects(a1, a2) and v where v is modified in a1, Affects is tested by varying the following features of a SIMPLE code:

- StmtType of stmts between a1 and a2 (e.g. assign, if, while, calls)
- When and where v is modified again in the control flow path between a1 and a2

We then build up our test cases in the order of increasing difficulty.

 <pre> graph TD     a1((a1)) --&gt; box[ ]     box --&gt; a2((a2))     style box fill:#ccc,stroke:#333,stroke-width:1px </pre>	<p><b>Case: Assign statements only</b></p> <p>The simplest test case. Within the assignment statements between a1 and a2, we check for the following 2 cases:</p> <ol style="list-style-type: none"> <li>1. v is modified within the grey box</li> <li>2. v is not modified within the grey box</li> </ol>
---	--

	<p><b>Case: Assignment statements with if-then statement(s)</b></p> <p>Cases where v may be modified by statements before a2:</p> <ol style="list-style-type: none"> <li>1. v is not modified in either the “then-block” or the “else-block”</li> <li>2. v is only modified in only the “then-block” or only the “else-block”</li> <li>3. v is modified in both the “then-block” and the “else-block”</li> </ol> <p>For each of the above cases, we also have to vary the positions of a1 and a2:</p> <ol style="list-style-type: none"> <li>1. a1 comes before the if node: <ol style="list-style-type: none"> <li>a. a2 is either in the “if-block” or</li> <li>b. a2 is in the “then-block” or</li> <li>c. a2 is after the endif node</li> </ol> </li> <li>2. a1 is either in the “if-block” or “then-block”: <ol style="list-style-type: none"> <li>a. a2 is in the other “if-block” or “then-block”</li> <li>b. a2 is after the endif node</li> </ol> </li> </ol>
	<p><b>Case: Assignment statements with with statement(s)</b></p> <p>A single while loop by itself will have 2 straightforward test cases:</p> <ol style="list-style-type: none"> <li>1. v is modified within the grey box</li> <li>2. v is not modified within the grey box</li> </ol> <p>For each of the above cases, we would also have to consider the positions of a2:</p> <ol style="list-style-type: none"> <li>1. a2 is within the while statement</li> <li>2. a2 is outside the while statement</li> </ol>

Subsequent test cases will involve more complex combinations of control flow paths. For example, nested container statements. These are crucial for checking for Affects\*.

Before testing Affects, we set up dummy data within components it is dependent on. An example of this set up is shown below:

```
void AffectsTest::MimicCodeWithAssignIf() {
    // 1. a = 6
    // 2. if b then { // a only modified in one
    // 3.     a = 3
    // 4.     k = a }
    //     else {
    // 5.     k = x
    // 6.     k = a }
    // 7. a = a + 2
    // 8. if b then { // a modified in both
    // 9.     a = 3
    // 10.    k = a }
    //     else {
    // 11.    a = 2
    // 12.    k = a }
    // 13.    a = a + 2
    // 14.    if b then { // a modified in none
    // 15.    k = a }
    //     else {
    // 16.    h = a }
    // 17.    x = a

    StmtTypeTable::Insert(1, ASSIGN);
    Modifies::SetStmtModifiesVar(1, VarTable::InsertVar("a"));
    Next::SetNext(1, 2);
    Follows::SetFollows(1, 2);

    StmtTypeTable::Insert(2, IF);
    Uses::SetStmtUsesVar(2, VarTable::InsertVar("b"));
    Modifies::SetStmtModifiesVar(2, VarTable::InsertVar("a"));
    Modifies::SetStmtModifiesVar(2, VarTable::InsertVar("k"));
    Uses::SetStmtUsesVar(2, VarTable::InsertVar("a"));
    Uses::SetStmtUsesVar(2, VarTable::InsertVar("x"));
    Next::SetNext(2, 3);
    Next::SetNext(2, 5);
    Follows::SetFollows(2, 7);
    ...
    StmtTypeTable::Insert(17, ASSIGN);
    Modifies::SetStmtModifiesVar(11, VarTable::InsertVar("x"));
    Uses::SetStmtUsesVar(17, VarTable::InsertVar("a"));

    cout << "\nSIMPLE-code-with-assign-if successfully created\n";
}
```

### PKB: Test 1 (Affects)

```
void AffectsTest::TestIsAffects() {
    ClearAllData();
    MimicCodeWithAssignOnly();
    ...

    ClearAllData();
    MimicCodeWithAssignWhile();
    ...

    ClearAllData();
    MimicCodeWithAssignIf();

    // var only modified in one of the 2 stmtLsts
    CPPUNIT_ASSERT(!Affects::IsAffects(1, 4));
    CPPUNIT_ASSERT(Affects::IsAffects(1, 6));
    CPPUNIT_ASSERT(Affects::IsAffects(1, 7));

    // var is modified in both stmtLsts
    CPPUNIT_ASSERT(!Affects::IsAffects(7, 10));
    CPPUNIT_ASSERT(!Affects::IsAffects(7, 12));
    CPPUNIT_ASSERT(!Affects::IsAffects(7, 13));

    // var is not modified in both stmtLsts
    CPPUNIT_ASSERT(Affects::IsAffects(13, 15));
    CPPUNIT_ASSERT(Affects::IsAffects(13, 16));
    CPPUNIT_ASSERT(Affects::IsAffects(13, 17));

    // misc odd cases
    CPPUNIT_ASSERT(!Affects::IsAffects(0, 1));
    CPPUNIT_ASSERT(!Affects::IsAffects(13, 18));
}
```

**Test Purpose:** Test if IsAffects() is able to return the correct result in various cases that have accounted for when optimising Affects.

**Required Test Inputs:** Dummy data in classes Modifies, Uses, Next, Follows and StmtTypeTable.

**Expected Test Result:** All assertions passed and no errors thrown.

### PKB: Test 2 (Affects\*)

```
void AffectsTest::TestIsAffectsT() {
    ClearAllData();
    MimicCodeWithAssignOnly();

    CPPUNIT_ASSERT(Affects::IsAffectsT(1, 3));
    CPPUNIT_ASSERT(Affects::IsAffectsT(1, 4));
    CPPUNIT_ASSERT(Affects::IsAffectsT(1, 5));
}
```

```
        CPPUNIT_ASSERT(!Affects::IsAffectsT(3, 5));  
        ...  
    }
```

**Test Purpose:** Testing if IsAffectsT() returns the correct results for a basic source code.

**Required Test Inputs:** Dummy data in classes Modifies, Uses, Next, Follows and StmtTypeTable.

**Expected Test Result:** All assertions passed and no errors thrown.



## Validation testing

### Testing tuples

Possible cases to consider for tuples:

1. Number of variables in a tuple
2. Similarity and differences between tuples in terms of:
  - a. Type of entity (e.g. stmt, assign, variable...etc)
  - b. Their relations to each other (if any) (e.g. Modifies(a1, a2))

#### Test 1

```
query = "assign a1,a2;while w;variable v;Select <a1,w,v> such that
Parent(w,a1) and Modifies(a1,v) pattern a2("x",_) with a1=a2";

std::list<std::string> expectedResult;
expectedResult.push_back("5 3 y");
expectedResult.push_back("6 3 y");

std::list<std::string> actualResult = QueryProcessor::ProcessQuery(query);
CPPUNIT_ASSERT(QueryProcessor::IsEqual(actualResult,expectedResult));
```

**Test Purpose:** To test whether select tuple works.

**Required Test Inputs:** Test query.

**Expected Test Result:** Correct output for Autotester.

#### Test 2

```
query = "assign a1,a2,a3;while w1,w2;variable v;Select <a1,a2,a3> such
that Parent(w1,a1) and Parent*(w2,a2) and Modifies(a1,v) pattern
a2("x",_) with a1=a2 and a3=10";

std::list<std::string> expectedResult;
expectedResult.push_back("5 3 10");

std::list<std::string> actualResult =
QueryProcessor::ProcessQuery(query);
CPPUNIT_ASSERT(QueryProcessor::IsEqual(actualResult,expectedResult));
```

**Test Purpose:** Test select tuple result with same synonym type.

**Required Test Inputs:** Test query.

**Expected Test Result:** Correct output for Autotester.

## 8. Discussion

### Areas that can be improved in the next iteration

#### Using Boost SL to implement bitvector

The major limitation of the STL's `vector<bool>` class is its inability to perform bitwise operations. This would not allow us to quickly check if 2 sets of variables contain the same values, which will be a disadvantage when it comes to optimising Affects. Therefore, the better alternative would be to consider using an external library.

Boost SL provides a `dynamic_bitset` class which will allow us to do bitwise operations while maintaining the ability for the vector to grow and shrink during runtime. If possible, we will look at ways to implement Boost's `dynamic_bitset` class to replace our existing `vector<bool>` classes.

#### Further optimisation of PKB components

PKB components that have been precomputed have already been optimised using bit vectors. Parts of the PKB that are currently computed on the fly can now find means of implementing bitwise operations wherever possible. Additionally, Affects is rooted deeply in optimal substructure and recurring subproblems, and as such dynamic programming can be extended across more Affects computations.