CS3201 Iteration I Report

Team 09

Filbert, Lim Jie, Niveetha, Wei Jie, Vivienne

Content

Abstract

1. Development Plan
2. Scope
3. SPA Design
    3.1. Overview
    3.2. Design of the SPA Components
        3.2.1. SPA Front-end Design
        3.2.2. PKB Design
        3.2.3. Query Processing
    3.3. Component Interactions
4. Documentation and Coding Standards
5. Testing
    5.1. Test Plan
        5.1.1. Unit Testing
        5.1.2. System (Validation) Testing
6. Discussion
7. Documentation of Abstract APIs
    7.1. PKB
    7.2. SynonymEntityPair
    7.3. QueryValidator
    7.4. Relationship
    7.5. RelationshipTable
    7.6. QueryElement
    7.7. QueryStatement
    7.8. QueryElement (object)
    7.9. QueryEval

Abstract

The Static Program Analyser (SPA) is a program that accepts code in a language code SIMPLE, and is able to parse through queries about the input program and respond to them. This report documents our approach to creating this program.

1. Development Plan

As there are four components in the SPA, each member handled a specific component, with two on PKB which could avoid merge conflict if worked on together. Once assigned to a component, each member performed full implementation, together with testing of the component. The development cycle started with us coming together to understand the type of data that will pass in and out of each component, and deciding on an API to facilitate interaction between the components. As a team, we also had to design the data structure of our PKB, as the parser and query evaluator heavily depended on this decision. With these two aspects of the project settled, we began work on our components. This iteration will end with all components passing unit testing, integration, and finally validation testing.

	PKB Design	PKB API	PQL API	PQL Design	PKB Parser	PKB Parser	Unit Test	Integration Test
Filbert		*	*	*				*
Lim Jie	*	*				*	*	*
Niveetha	*	*					*	*
Vivienne						*		*
Wei Jie			*	*	*		*	*

2. Scope

The basic requirements of SPA in iteration 1 include parsing a single procedure, parsing queries, and responding to these queries. Our SPA parses an input program line-by-line, and populates the tables implemented in the PKB to be stored. Our SPA can answer Follows, Parent, Modifies and Uses queries, and recognise patterns in the input program. The PQL is able to split queries into its various types, and passes it to the query evaluator, where it will determine which functions should be called from the PKB to respond to these queries. The PKB's API provides methods for the query evaluator to obtain specific data about the program (for example, what statements follow statement 5, etc.). Once the PKB responds to the query evaluator, the evaluator determines the correct results and outputs the value.

3. SPA Design

3.1. Overview

Our SPA components include the program parser, PKB, query parser, and the query evaluator. The parser dissects the program lines based on its type (while, if, etc.), and calls PKB setter functions accordingly. The PKB is implemented using tables, and stores instead of the program in AST form, answers to possible queries from the PQL. A substantial amount of query processing is done in the PKB, and the query evaluator calls for pre-computed results, and outputs values according to its input query; the query evaluator breaks the query into multiple parts and queries the PKB, putting together sets of responses and finally output the correct set of result. The PQL aids the query evaluator with this process, parsing the query as clauses to be received by the evaluator, and later identified to make appropriate calls to the PKB and output the final result.

3.2. Design of the SPA Components

3.2.1. SPA Front-end Design

The parser takes in the code and extracts the parts of the code that is relevant to populating the PKB. Here, we are also assuming we are dealing with one Procedure only. A lineCounter variable is in place to track the number of lines so we can read the relevant information to the PKB. We have have also set up a vector<int> Parent so we can keep track of what is the immediate parent to a certain line.

To begin with, it reads in the first line of the code and verifies if it is a valid procedure. If it is, it reads the next line. Nesting level would increase. The goal here is to make sure the program can answer queries such as "Select w such that Parent(w, 7)" etc. We have 4 types of relationships to note: Follows, Parent, Modifies, Uses. For every line, we take note of the nestLevel (variable is as such) - if the nestLevel is more than 1 the last line number in the parent is matched with the current line for a parent pair. lineCounter increases for every line.

If it is an assignment statement: (a) The variable on the left side of the equation is being modified, so we call setModifies from the PKB which notes the line of which the variable is being modified, or (b) if the previous line and the current line are on the same nesting level (boolean isSameLevel variable has to be true), we call the setFollows from the PKB to read in both line numbers. If it is not, the last element from the Parent vector is used, and (c) when it comes to reading in the assignment

statement itself, the constants are read into a constants table via addConstant from PKB. The variables will also be read into a variable table via the addVariable from the PKB, and since all the variables on that line is being used, we call setUses from PKB.

If it is an while/if statement, a variable has been used on that line, so we call setUses from the PKB which takes notes of the variable being used on that specific line. While/if are the start of a nesting statement, so the line they are on is added to the Parser Parent vector, and isSameLevel becomes false. If it is an else statement, it has no statement number. We decrease the lineCounter variable and continue. Lastly, the parser checks for the closing brackets in that line. Nesting level decreases at the end of each closing bracket, so nestLevel will decrease. isSameLevel would also be false. The nestLevel is mainly to check if the number of the opening and closing brackets are the same in the code, and if it's not Parser returns false.

For pattern identification, we created a method to insert brackets into the statement, so as to give it the form it would have in an AST; these pattern variations are stored for later use, and accessed by the evaluator during querying.

3.2.2. PKB Design

The PKB is designed after a black box, where insertion of values comes only from the parsed in code, and results are obtained by the query evaluator with only knowledge of the API.

The PKB is first accessed by the parser. The parser calls the PKB's set functions and stores all variations of Follows, Parent, Modifies and Uses in its respective tables. The PKB contains a varIndexTable and a procIndexTable (which is private to its class) – the index table gets the index of any variable in the program, or if it does not have one, creates and returns it. It also has tables that identify statement types (statement types being assign, if-else and while). The PKB is also able to return all statements that are of the various statement types, all variables/constants in the program, and all statement numbers.

Our PKB methods was designed to reduce setter calls from parser, for example: in the setModifies method, for setModifies(3, "z"), in addition to adding statement 3 modifies variable z in our ModifiesTable, we also call the function getParent(3), and

setModifies((iterate through all parents), “z”);. For functions such as modifies, uses, etc., the modifiedBy, usedBy functions and implicitly called within the functions itself, freeing the parser from these calls.

The PKB provides conversion from variable/procedure names to an index values. This allows for easier method calling and programming in the PKB – the parser/evaluator calls a method of the PKB, the PKB converts the parameters in the function to an index form, and calls the function accordingly. This removes the task of indexing from other components (namely the parser and evaluator, from both having indexing tables, to only the PKB); this is a more logical design implementation as the index table is only required for the manipulation of data, which is done in the PKB.

3.2.3. Query Processing

During query processing, if a declaration is encountered, the associated entity and synonym used are placed in the SynonymEntityPair, while ensuring that the entity declared is valid. If a “Select” is encountered, the associated synonym is captured and validated by checking with the declaration. If a “such that” is encountered, its relationship(e.g. Follows) is captured while making sure that this relationship is valid by checking with the relationshipTable; this is done by checking if it has the correct types of arguments. If a “pattern” is encountered, its entity (e.g. a) is captured and validated. Later, further check that the arguments are valid by making sure that it passes a certain pattern.

As queries are input into the PQL, the arguments’ relationships are stored – this is done through the Relationship class. This relationship is validated using the RelationshipTable class, which contains all possible valid relationships and returns a Boolean upon invocation.

The query evaluator takes in the PKB and a query statement which consists of queries as its input. The module will extract out information from the statement with the help of PQL processing.

By calling the relevant API’s from PQL, the query statement is broken down into select, such that and pattern clauses. The program begins to evaluate the select clause to determine the common synonym and the attribute type for the query (an

integer vector which represents statement values or string vector containing variables).

Next, it will begin to evaluate the select clause. A switch implementation is used to determine the procedure to execute depending on the entity attribute of the synonym. There are two type of vectors to fulfill this objective, integer vector: statement, assign, while, and string vector: variable, constant

After analysing the select clause, the program proceeds to evaluate the Such That clause. The function first determines whether the argument 1 is an integer. Next, it will check whether there exist a common synonym in the clause which is encoded in the variable comEval. A value of 0 represents that both argument 1 and 2 are synonyms which are different from the select clause.

Case: Modifies and Uses:
The function calls for the appropriate API depending on comEval variable.

Case: Follows, FollowsStar, Parent, ParentStar
Checks whether both arguments are integers, and check the validity of the clause by comparing the result of the API against the 2nd argument. Otherwise, calls the appropriate

With regards to pattern matching, extract the processed expression, synonym and entity from PQL. If both argument 1 and 2 are wildcards or the entity is not a common synonym, the function checks that there exists at least one statement with an expression. If the common synonym is the entity, it will return statement values. If the common synonym is argument 1, it will return string vector of variables.

All Argument 2 (pattern expressions), are processed in bracket form so that the query evaluator can compare against the expression stored in the PKB easily.
For example: x = y+z*a – b, the expression will be stored as ((y+(z*a))-b) in the PKB.

The pattern expression argument 2 is classified under 3 different categories:
wildcard:
substring: uses find() from standard library to check if the query expression is a substring of the assignment statement.

exact: compares the given query expression to the PKB expression.

Therefore, given a query Pattern a(_,_"y+z"_),
The function will evaluate (y+z) against ((y+(z*a))-b) which will return an empty string since it is not found.

If there exist at least one pattern which is satisfied, the statement number/variable name will be stored and the variableClausePattern will be set to true. For ease of access to variables and its statement number, the information is stored in a vector<tuple<int,string>>.

After having evaluated each individual clause, the module will compare the select and such that clause , taking the intersection of answers where appropriate as queryAnswerInt or queryAnswerString. Finally it will take the intersection of queryAnswer and patternClause to obtain the final answer for the given queryStatement. The module then returns this value as a vector string. If the answer is a vector of integers, it is converted to a vector string. The query answer will be passed on to the UI.

3.3. Component Interactions

The SPA begins at the PKB parser, where it dissects the input program lines and populates the PKB (components interaction seen between parser and PKB). The PQL takes in queries, and there dissects the queries into entities to be passed to the query evaluator (components interaction between PQL and evaluator). The evaluator determines which of the PKB's API it would need to call to respond to the query (components interaction between PQL and evaluator). Once the evaluator determines the result of the query, it outputs them.

4. Documentation and Coding Standards

As version control is simple and easy to use for a large team, documentation was done using the same version control system we used for our project – Git. All documents related to this project were managed using version control for simplicity.

The SPA API was constructed by the members who worked on the PKB, and commenting was done in the PKB header file to aid components interacting with it with function calls (namely the parser and query evaluator). We've adopted the conventional naming standard, camel case, and for the parameters such as (int

statement), we've chosen to name statement as "s", if there is only one statement in the function, and "s1", "s2", … for more than one statements, etc.

5. Testing

The SPA has multiple components that rely upon each other to function. As the SPA components were created independently of one another, it was vital that each component performed as it should; all components were created with the assumption that the others were performing as required. To test components independently, it was essential that all of us did unit testing before integrating these components. Unit testing, and AutoTester, provides a platform for us to continuously check if our code is running as programmed, and lets us know immediately if it does not.

5.1 Test Plan

For iteration 1, testing was mainly done on individual components (i.e. unit testing), and ensuring the methods were working as required. Each coder provided together with their code, unit tests to ensure that their component could be integrated with the others.

5.1.1. Unit Testing

An example of unit testing was checking for methods that take in string variables, performing conversion to be stored in index tables, and responding to call functions with the variable names (this requires conversion back from its index). Dummy values were used to run these processes, and by asserting the correct results, we were able to proof the component was accessing the right data structures, doing the correct manipulations, and returning the corresponding values.

5.1.2. System (Validation) Testing

Integration was performed on three components: PKB, query evaluator and PQL. System testing was done by manually populating PKB instead of parsing in a program. Queries were passed through the PQL, and its response is output by the query evaluator as a string.

6. Discussion

With a five-man team, the largest difficulty we faced was being held responsible for a certain component in the SPA. With a program parser, PKB, program evaluator and

PQL parser, each one of us had to handle one component alone, with the exception of the PKB, and understand the components interacting with it to make the appropriate API and function calls.

When we first designed our SPA implementation, we decided to store the entire program as an AST (linked list) and using traversal, obtain results for the query evaluator. We soon realised that storing the program in well-defined tables is much more advantageous in terms of efficiency with regards to results-calling - we can have pre-computed values as possible queries to the PKB can be easily created by us. This was a design decision, and one we stand by with the implementation of SPA.

The scale of this project slightly underestimated by us, and only as we started to implement the program did we realised the complexity and scale of the program. I do believe with more in-depth discussions on the implementation and with larger dissection of tasks (with shorter deadlines), we could have found the implementation of SPA to be less overwhelming. Better management with regards to watching over and enforcing the completion of these smaller tasks would have greatly helped in this iteration.

7. Documentation of Abstract APIs

7.1. PKB

vector<string> getAllVariables(); Returns all variables in PKB
vector<string> getAllConstants(); Returns all constants in PKB
vector<string> PKB::getAllProcedures(); Returns vector string of all procedures in PKB
void setFollows(**int** s1, **int** s2); Sets s2 to follow s1. Meaning: s2 > s1.
vector<int> getFollows(**int** statementNum); Gets the statment number that follows the input statement.
vector<int> getFollowedBy(**int** statementNum); Gets the statment number that is followed by the input statement.
vector<int> getFollowsStar(**int** statementNum); Gets the statment numbers that follows* the input statement.
vector<int> getFollowedByStar(**int** statementNum); Gets the statment numbers that is followed* by the input statement.
void setParent(**int** s1, **int** s2); Sets s1 to be the parent of s2. Meaning: s2 > s1.
vector<int> getParent(**int** statementNum); Gets the statment number that is the parent of the input statement.
vector<int> getChild(**int** statementNum); Gets the statment number that is the child of the input statement.
vector<int> getParentStar(**int** statementNum);

Gets the statment numbers that are the parent* of the input statement.
vector<int> getChildStar(**int** statementNum); Gets the statment numbers that are the child* of the input statement.
void setModifies(**int** s, **string** varName); Sets s to modifies varName. Implicitly set varName to be modifiedBy s and sets modifies for container statements.
void setProcModifies(**string** procName, **string** varName); Sets proc to modifies varName. Implicitly set varName to be modifiedBy proc.
vector<string> getModifies(**int** s); Gets the variables statement s modifies
vector<int> getModifiedBy(**string** varName); Gets the statements modified by variable varName
vector<string> getProcModifies(**string** procName); Gets the variables the procedure modifies
vector<string> getProcModifiedBy(**string** varName); Gets the procedures modified by variable varName
void setUses(**int** s, **string** varName); Sets s to uses varName. Implicitly set varName to be UsedBy s and sets uses for container statements.
void setProcUses(**string** procName, **string** varName); Sets proc to uses varName. Implicitly set varName to be UsedBy proc.
vector<string> getUses(**int** s); Gets the variables statement s uses
vector<int> getUsedBy(**string** varName); Gets the statements used by variable varName
vector<string> getProcUses(**string** procName); Gets the variables the procedure uses
vector<string> getProcUsedBy(**string** varName); Gets the procedures used by variable varName
void setStatementType(**int** statementNum, **int** type); Sets the statement type of each line of the program
vector<int> getWhile(); Gets all statement numbers of While statements
vector<int> getAssign(); Gets all Assign numbers of While statements
vector<int> getIf(); Gets all statement numbers of Is statements
vector<int> getAllStmt(); Gets all statement numbers in program
void addConstant(string c); Add a Constant to PKB
void addPattern(**int** StatementNum, **string** leftVariable, **string** rightExpression); Adds a pattern to PKB
vector<tuple<int, string>> getPattern(**string** varName); Gets all the statement number and expressions that appears for a variable

7.2. SynonymEntityPair

vector<string> getSynonymList() Return a vector<string> of synonyms is associated with a particular entity
string getEntity() Returns a string of the entity that is in the SynoymAndEntityPair
SynonymEntityPair(**string**, vector<string>); Creates a synonym entity pair.

7.3. QueryValidator

bool parseInput(**string** str) Takes in a string argument i.e. query. Returns true if query is valid, else false
bool isEntityAndSynonym(**string** str); Takes in a string str argument which contains of the initial part of queried synonym. Returns true if the str is valid design entity, else false.
bool isValidSuchThat(**string** str, **string** syn); Takes in 2 string arguments and check if the such that clause is valid. Returns true if valid, else false.
bool isValidEntity(**string**); Takes in a string argument and check if entity is valid. Returns true if valid, else false.
bool isVariableArg1(**string** arg1); Checks if input argument is a variable. Returns true if valid, else false.
bool isSubstringArg2(**string** arg2); Checks if input argument is a substring. Returns true if valid, else false.
bool isWildcard(**string** arg); Checks if argument is a wildcard. Returns true if valid, else false.
bool isExactString(**string** arg2); Checks if input argument is an exact string match for pattern. Returns true if valid, else false.
bool isValidQueryLine(**string** selectString); Checks if input string (program line) is a valid query.
void startParsing(**string** str); Takes in a string argument and starts passing the string argument as a query
string changeLowerCase(**string** str); Takes in a string and changes it to lower case and returns the string in lower case.
string removeSymobls(**string** str, **string** symbolToRemove); Takes in 2 string arguments and remove all the existing string specified in 2nd argument in 1st argument. Returns the 1st argument without the symbolsToRemove.
vector<string> split(**vector<string>** vectorToSplit, **string** strToSplitWith); Takes in 2 arguments and split the string accordingly to the occurences of strToSplitWith and places them into a vector<string> that is to be returned.

7.4. Relationship

vector<string> getArg1(); Returns the vector of design entities of the first argument of a relationship
vector<string> getArg2(); Returns the vector of design entities of the second argument of a relationship

7.5. RelationshipTable

bool isValidArg(**string** rel, **string** type, **int** numArg) Takes in a relationship argument and the type of design entity the argument is, to be used for public calls, calls isValidArg1 or isValidArg2 accordingly. Returns true if the type of design entity exists in the relationshipTable, else false
bool isRelationshipExists(**string** rel) Takes in a string argument rel and checks if this string is a key in the unordered_map(relationshipTable). Returns true it exists, else false.

7.6. QueryElement

string getSuchThatRel() Returns the type of relationship
string getSelectEntity() Returns the entity associated with synonym being queried in Select query.
string getSelectSynonym() Returns the synonym bring queried in the Selct query.
string getSuchThatArg1() Returns the first argument that is being queried on in a relationship
string getSuchThatArg2() Returns the second argument that is being queried on in a relationship
string getSuchThatArg1Type() Returns the first argument type that is being queried on in a relationship
string getSuchThatArg2Type() Returns the second argument type that is being queried on in a relationship
string getPatternArg1() Returns the first argument of a pattern
string getPatternArg2() Returns the second argument of a pattern
string getPatternEntity() Returns the entity of the pattern
string getPatternSynonym() Returns the synonym of the pattern
string getPatternArg1Type() Returns the first argument type of a pattern
string getPatternArg2Type() Returns the second argument type of a pattern

7.7. QueryStatement

vector<QueryElement> getSuchThatQueryElement() Returns the vector<QueryElement> that is associated with such that.
vector<QueryElement> getPatternQueryElement() Returns the vector<Queryelement> that is associated with pattern
vector<SynonymEntityPair> getSynonymEntityList() Returns the synonym and Entities that were declared in a vector<SynonymEntityPair>
void addSelectQuery(**QueryElement**) Adds the select portion (Without clauses) of a query statement as a queryElement into the object
void addSuchThatQuery(**QueryElement**); Adds the such that portion (without clauses) of a query statement as a queryElement into the object
void addPatternQuery(**QueryElement**); Adds the pattern portion (without clauses) of a query statement as a queryElement into the object

7.8. QueryEval

string QueryEval(**PKB, qs**) Takes in PKB of program and queries to be evaluated.
vector<string> runQueryEval() Runs query evaluator on query and returns a vector<string> type of the result.