CS3201 Iteration I Report

*Team 09*

Filbert, Lim Jie, Niveetha, Wei Jie, Vivienne

Content

Abstract

1. Development Plan
2. Scope
3. SPA Design
   1. Overview
   2. Design of the SPA Components
      1. SPA Front-end Design
      2. PKB Design
      3. Query Processing
   3. Component Interactions
4. Documentation and Coding Standards
5. Testing
   1. Test Plan
      1. Unit Testing
      2. System (Validation) Testing
6. Discussion
7. Documentation of Abstract APIs
   1. PKB

**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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Filbert | Lim Jie | Niveetha | Wei Jie | Vivienne |
| Program Parser |  | \* | \* |  | \* |
| PKB |  | \* | \* |  |  |
| PQL Evaluator | \* |  |  |  |  |
| PQL |  |  |  | \* |  |
| Report |  |  | \* |  |  |

1. **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.

1. **SPA Design**
   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.

* 1. Design of the SPA Components
     1. SPA Front-end Design

*Describe how you parse and validate SIMPLE source code. Explain when and how your PKB is populated by Parser / Design Extractor. b. PKB design ‐ Explain the structure of your PKB, as well as when and how*

* + 1. 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.

* + 1. Query Processing

*Explain your query validation. A sample query validation rule is: "Check if all relationships have the correct number and types of arguments". DO NOT provide procedural description (pseudocode) of how Query Pre‐processor checks the rules. Very briefly, describe how you encode validation rules, and how you perform query validation. Describe your design and implementation of Query Evaluator ‐ describe the data representation for program queries (query tree) and how your Query Evaluator works.*

* 1. Component Interactions

*Include any UML diagram that you have found useful. For each diagram, explain how you have used it (e.g., in project planning, communication, test planning or in other situations), and comment on the value it has added to your project.*

1. **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.

1. **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.

* + 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.

* + 1. System (Validation) Testing

-

1. **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.

1. **Documentation of Abstract APIs**
   1. PKB

|  |
| --- |
| **vector<string>** getAllVariables();  Returns all variables in PKB |
| **vector<string>** getAllConstants();  Returns all constants 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, **vector<int>** parentStarOfStmt);  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, **vector<int>** parentStarOfStmt);  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 |

* 1. ProcTable
  2. Follows, Follows\*
  3. Parent, Parent\*
  4. Modifies, …