National University of Singapore

CS3203 Software Engineering Project

AY21/22 Semester 1

Project Report

Team 35

Consultation Hours: Mondays 5-6pm

Tutor: Wang Yuting

Team Members	Student No.	Email
Lim Junxue	A0204757B	junxue.lim@u.nus.edu
Max Ng Kai Shen	A0206101E	max.ng@u.nus.edu
Tan Wei Jie	A0202017B	t_weijie@u.nus.edu
Hemanshu Gandhi	A0180329E	hemanshu.gandhi@u.nus.edu
Oliver Cheok	A0136236J	oliver.cheok@u.nus.edu
Ritesh Kumar	A0201829H	ritesh.k@u.nus.edu

Abstract

Our Static Program Analyzer (SPA) is an outcome of extensive deliberation, planning and concerted programming effort — something we are proud to present. For this mini-SPA, we would like to bring to light our own architectural choices that value-add to the ones prescribed by the module, elaborate on other fascinating design approaches taken and our own spin on some of the common algorithms used in SPAs.

We designed our architecture with two primary goals: to maximize the benefits of Object-Oriented Programming (OOP) and to keep data flow straightforward. To maximize the gift of OOP, we identified intuitive design patterns and created purposeful subcomponents in our front end that are more granular than what was prescribed. This allows our front-end subroutines to form clear pipelines that parse the input SIMPLE program in a single pass. As part of a best-effort approach, every data transformation value-adds to the information gleaned so far. To keep data flow straightforward, we ensure that there's always a single direction of data flow between one component to the next in the pipeline. Additionally, we made it such that the Program Knowledge Base (PKB) is written to for only one time because this ensures that PKB focuses only on maximizing the speed of reading from it, as a data store should.

Our code base is extensible and future-ready for requirement changes and our adherence to various design principles and patterns gives us this confidence. For example, we passed wrapper objects between larger components to easily adhere to clearly defined APIs that can be quickly modified should their internal data representations need any modification. We also minimized the code base by sharing common data formats and utilities among the Source Processor as well as the Query Processing Subsystem, a benefit of following the DRY principle.

Our multiple subcomponents are easily swappable and allows us to use adaptation of common algorithms flexibly. For example, both our front-end and back-end parsers use a Recursive Descent Algorithm but at the same time, we also employ an adaptation of the Shunting Yard algorithm to create postfix strings that represent assignment expressions. Depth First Searching is utilized when creating entities and relationships as well. This flexibility in use of algorithms was possible due to the discrete subcomponents mentioned earlier.

We have ensured that Optimizations are not an afterthought either. Low-hanging fruits such as using string representations and string-matching (of postfix expressions), using maps for direct O(1) time memory access were taken. Additionally, we employed a grouping algorithm in the backend that groups clauses based on common synonyms. The component also supports common heuristics and determines an efficient order of dispatching queries to the PKB. We used a tables-based approach for query evaluation, to mimic the set operations that databases do

and capitalized on space by creating helpful auxiliary data structures for quick lookups.

Finally, we added some convenience features for users and developers alike. There is a singleton Logger that is easily configurable to output log statements via macros and output messages into either the console or a file of choice or both. Error handling is done with care so that if the program terminates, the user is never left in the dark as to why. Convenience scripts in our CI pipeline make testing a hassle-free process that is a boon to developers.

We propose two extensions to SPA, namely the addition of Dominates and Postdominates relationship as well as the introduction of cyclomatic complexity analysis (which measures code complexity). These proposals are discussed in more detail later in the report.

We at Team 35 hope that this document proves to be an interesting read.

Contents

1. Static Program Analyzer (SPA) Design

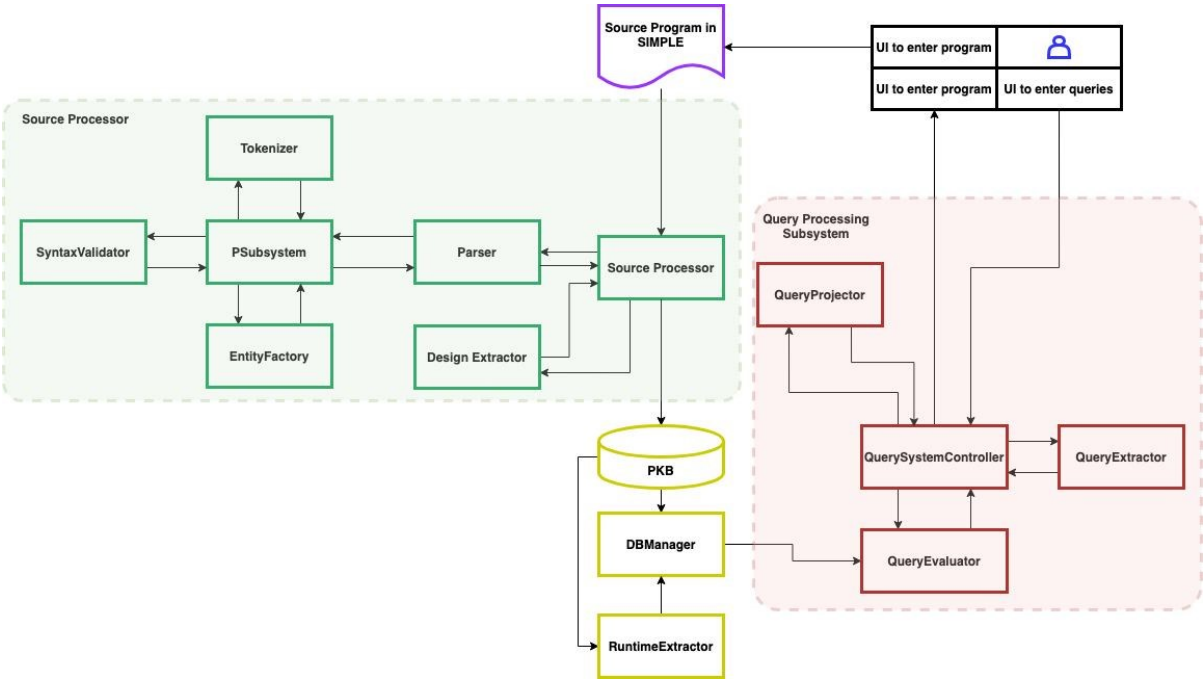


Figure 1 Architecture Design

As an overview, our SPA design has larger components which take inspiration from common Software Projects hence there are many recognizable roles in our components. In general, we have also adapted common software design patterns used in Compiler Design since many of the required steps in a SPA share similar logic with compilers:

1. **Source Processor:**
   The Source processor is analogous to a front-end component since it interacts directly with the client and consumes the input source code supplied by the Client. Its role is to carry out steps like the Analysis phase of compilers, where syntax is parsed and validated, and the source code is converted into abstract data formats and other lookup tables are populated during the Lexical Analysis phase. These data models / entities are eventually stored in the PKB via a single write.
2. **Program Knowledge Base (PKB):**
   This is a read only store of data that exists to supply query results tailored to read the table-like data structures that the PKB keeps. It mimics database operations and hence is responsible for data representation that is done in a query-efficient manner.
3. **Query Processor:**
   This component is analogous to the backend of software since it deals with data retrieval and analysis. The Query processor is responsible for the processing and validation of queries as well as the evaluation and repackaging of data read from PKB in an efficient way that handles users' queries.

4. **DB Manager:**
   A proxy to the PKB, this component mirrors the read operations that can be done on the PKB and can route the query through the RuntimeExtractor for runtime-specific relationships like Affects.
5. **Runtime Extractor:**
   Handles runtime-specific relationships and maintains a per-relationship cache for speeding up evaluations within a single query.

1.1 Our SPA Vocabulary

This section establishes some common vocabulary which shall be used to elaborate on our SPA's implementation.

SPA Primitives

These classes create an abstraction of the data type they store, while enforcing its grammar syntax.

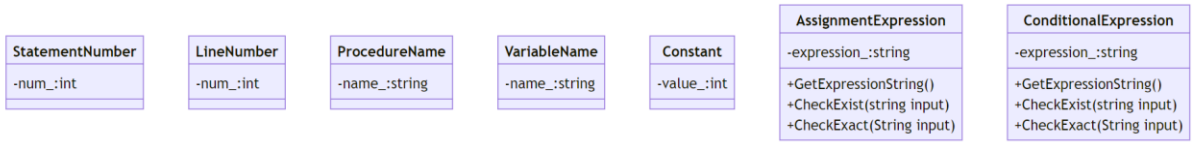


Figure 2 Primitive Types

Entities

Entities refer to the classes that contain information from the SIMPLE program.

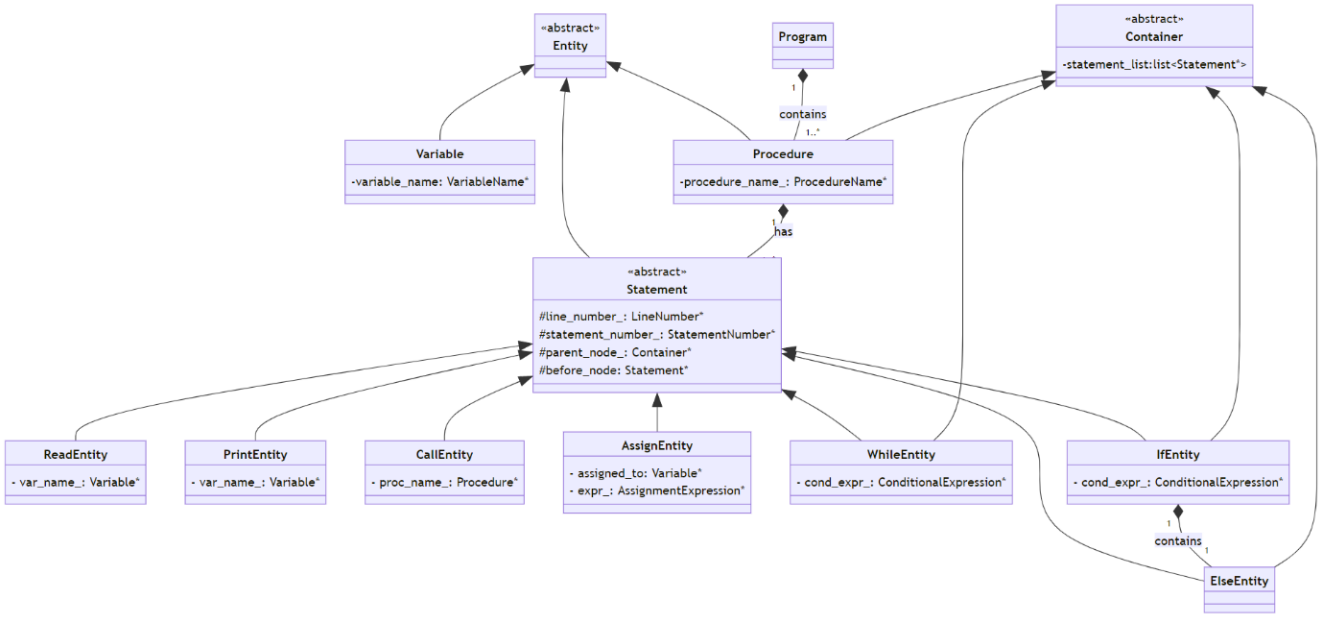


Figure 3 Entity Class Diagram

To prevent overcomplication of this class diagram, all functions of classes are abstracted as well as most of the relationships are stated as variable-style instead of

arrows relationship. The main purpose of this class diagram is to indicate the abstract classes and interfaces each Entity-type class inherits from.

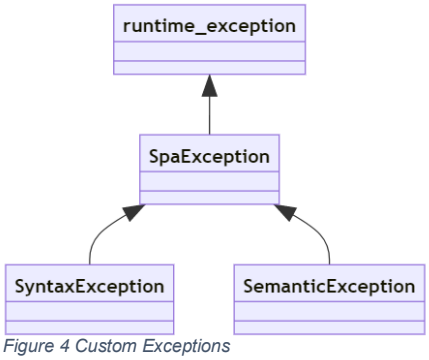
This class diagram is also appended in Appendix – D to increase readability.

Containers

Containers contains a list of Statements which are inside a pair of braces, only for 1 level of nesting. As depicted in the Entities class diagram, a Procedure, WhileEntity, IfEntity and ElseEntity can be Containers. When referring to Containers in this report, a Container statement refers to WhileEntity, IfEntity or ElseEntity.

Custom Exceptions

Custom exceptions are created to generate meaningful errors that are specific to this project. Our base exception class, `SpaException` is subclassed from `std::runtime_exception`.The other derived classes help identify whether the exception is due to Syntax of Semantic errors in the SIMPLE program & queries.



mple Exception Handling

Figure 4 Custom Exceptions

Our system ensures that all exceptions are handled gracefully. These custom exception classes are useful in categorizing the exceptions we throw and give us the opportunity to handle error logging specific to the error, for meaningful error messages to be displayed when these exceptions are thrown.

We created a global logger object that can be configured for file or stdin and other basic configurations; necessary log statements can be recorded during each run. This helps the SPA user to easily trace the bug and fix issues.

SPA PQL

All technical terms used according to the grammar rules in the SPA's PQL wiki will be followed in this report

1.2 Sample SIMPLE program

The following SIMPLE program shall be used for the rest of this report as a common reference to explain the inner workings of our SPA. The highlighted parts of the

SIMPLE program are **some** of the interesting aspects that we will elaborate later in the report.

Stmt #	SIMPLE Program Statement
	procedure SampleProc {
[01]	read x;
[02]	y = 2 * z;
[03]	while (a>=(1*(1/(10)))) {
[04]	while ((nice + 1) > 12) {
[05]	y = x * q - 5;
[06]	z = z - 1;
[07]	if (q != z) then {
[08]	x = 1 ;
	} else {
[09]	print x;
[10]	read x;
	}
[11]	i = x + j + z;
	}
	}
[12]	if (report <= 60) then {
[13]	print print;
	} else {
[14]	read pages;
	}
[15]	call Proc2;
	}
	procedure Proc2 {
[16]	call Proc3;
[17]	B = c;
	}
	Procedure Proc3 {
[18]	D = e;
	}

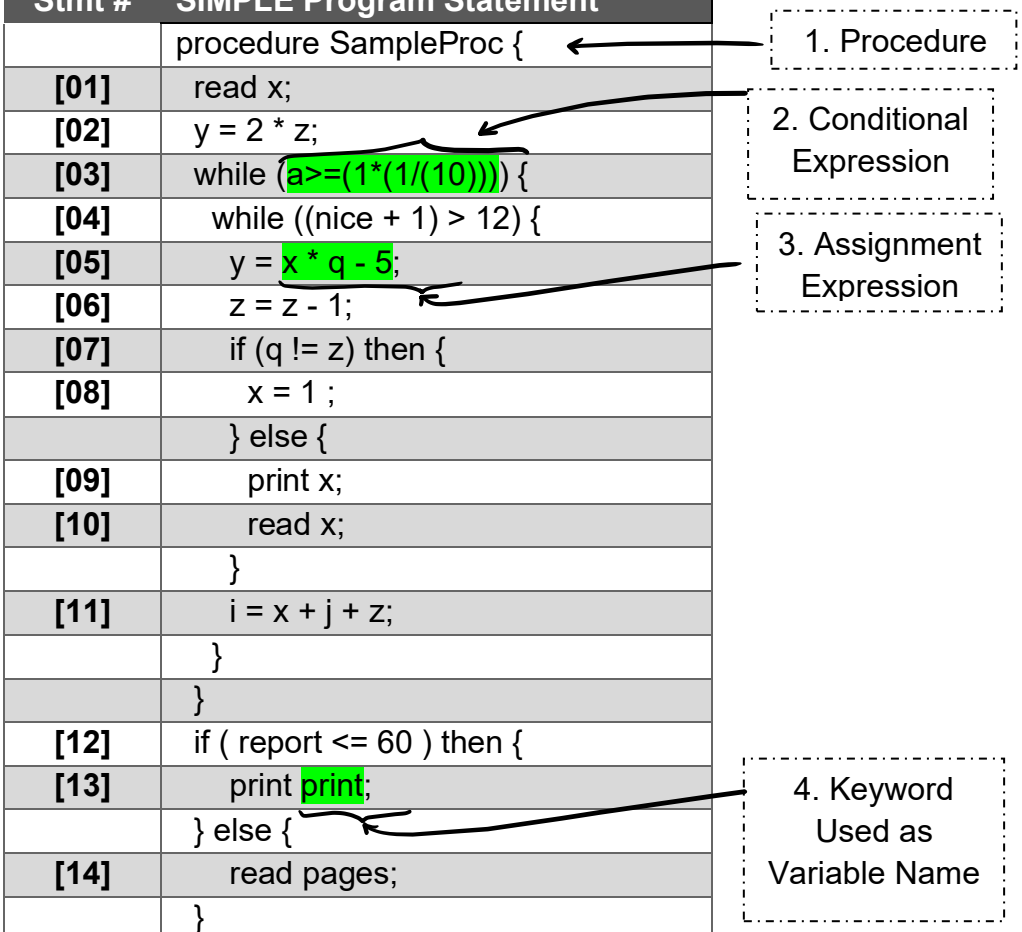


Table 1 Sample SIMPLE code

1.3 Source Processor

The parsing technique used by the Source Processor is modelled after commonly established parsing techniques. Intuitively, we went for a top-down, recursive descent parsing technique. The process is simplified by the separation of concerns between the Parser, which adapts the stack-based approach to parse the SIMPLE program across Statements, and the validation of grammar syntax of a Statement, handled by the Tokenizer and SyntaxValidator. This separation allows the

SyntaxValidator to fulfil the role of parsing the left-associative assignment Statements in this top-down parser.

The sequence diagram below illustrates how the data flows when the Source Processor parses the SIMPLE program.

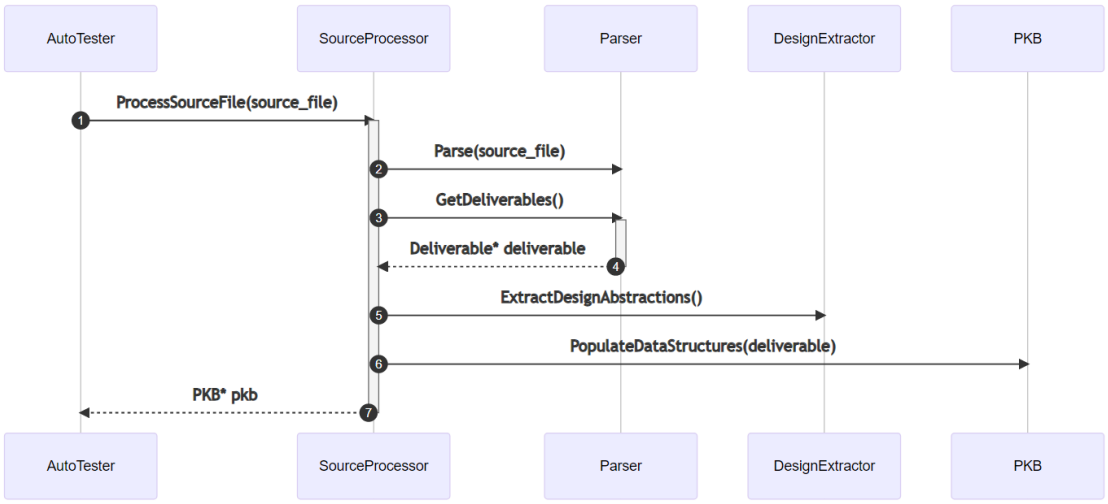


Figure 6 High Level Data Flow

Here, the Source Processor is a façade that provides an entry point into the process of parsing, creating abstract data representations in the process. It creates a pipeline with three steps, namely parsing, extracting design abstractions, and then populating the PKB.

1. Firstly, the Parser also acts as a façade that abstracts implementation details related to the lexical analysis, syntax validation and creation of entities.
2. Secondly, the Design Extractor carries out a deeper analysis of the relationships within the entities identified by the Parser, to extract transitive relationships.
3. Thirdly, the Parser passes a wrapper object, called a Deliverable, to the PKB in a single “write” operation. The Deliverable encapsulates all the necessary information required on the SIMPLE program, which functions as a Data Transfer Object.

We will go into the details of parsing and design extraction in the next 2 sections, Parsing and Design Extraction.

1.3.1 Parsing

Parsing handles data on a per-statement level and creates abstracted entities that represent that exact statement via objects. This mimics the lexical analysis phase of a generic compiler.

Let us use statement number 3 from our sample program to demonstrate the process. The overall process can be summarized as a pipeline of events where the output of each component is the input of the next:

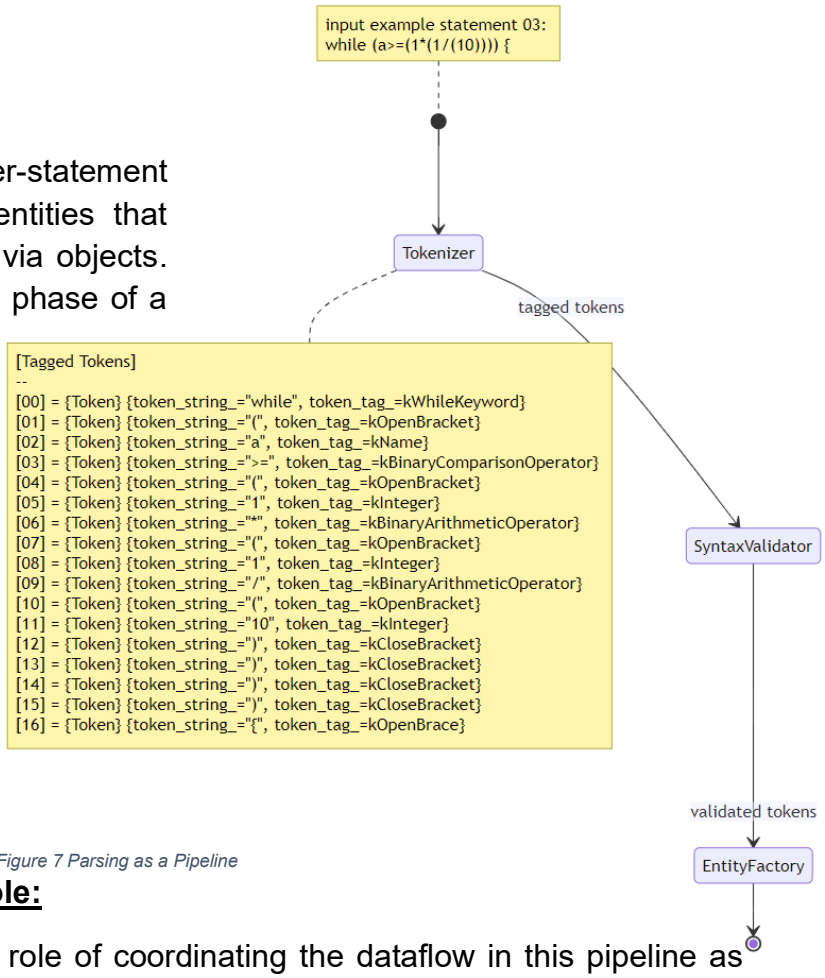


Figure 7 Parsing as a Pipeline

PSubsytem's Coordinating Role:

The PSubsystem plays the key role of coordinating the dataflow in this pipeline as described by the following sequence of actions:

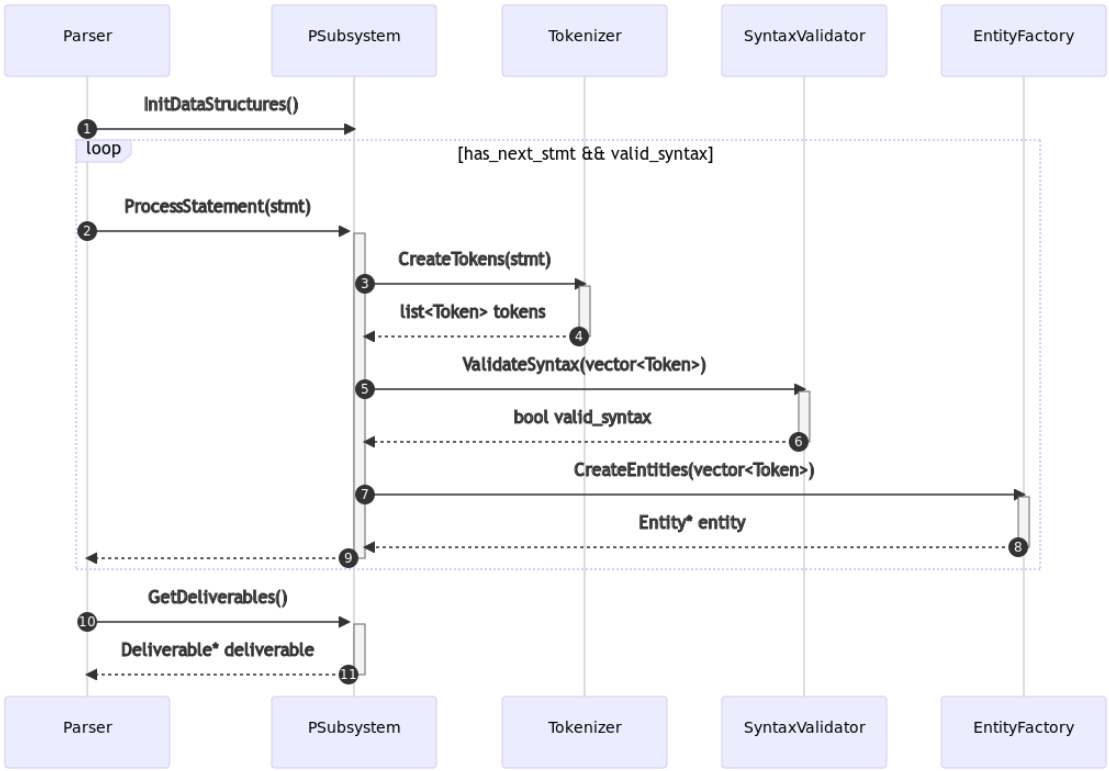


Figure 8 Parsing Sequence Diagram

Eventually, by relying on intermediate representations in the pipeline, the input source code is synthesized into the abstract entities and some basic relationships are represented in query-efficient lookup tables. The Tokenizer, SyntaxValidator and EntityFactory handle their responsibilities on a per-statement level as well.

Tokenizer

Taking inspiration from Parts of Speech Tagging, we represent each distinct string into a Token, where a string is tagged with its appropriate TokenTag (Figure 7) with the note on Tagged Tokens). Having a fixed set of TokenTags makes the processing of logic in subsequent steps of the pipeline much more manageable.

The tagging is done by regex string matching from a central RegexPatterns class which implicitly allows us to set syntax rules on a per-token level. For example, anything tagged as a name variable is guaranteed to follow the regex pattern of

`^[[:alpha:]]+([0-9]+|[[:alpha:]]+)*` so we automatically validate names to follow the concrete grammar specification that names must start with a letter.

Consider the need for a Look-Ahead Parser to support the use of keywords as synonyms/variable names, as shown in Statement # 13 in the SIMPLE Program (`print print`), where the first `print` is a keyword, and second `print` is a variable name. To achieve these functionalities, the parser shall verify if a tagged keyword is really an actual keyword by considering the other tokens and their relative positions in the same statement.

Syntax Validator

Enforcement of Concrete Grammar Syntax on a per statement-level is done by the SyntaxValidator, and the general Activity of how Syntax Validation is done:

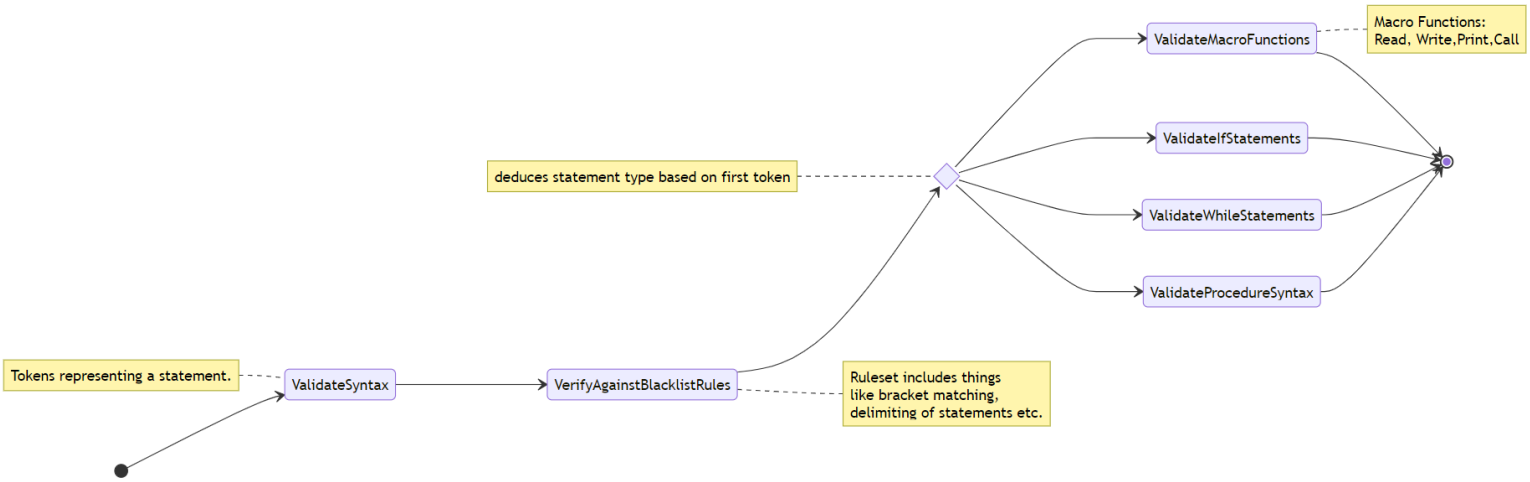


Figure 9 Syntax Validator Logic

The use of blacklist rules (for example, a non-container line's last character, if it's not "**;**", makes the line invalid) helps the Validation routine end early the soonest common SIMPLE code issues are detected, without having to enter the Validation functions.

Now, the Validation functions rely on a set of helper functions. These functions verify whether a contiguous range of tokens within a collection of tokens that represent the statement adhere to the Concrete Syntax Grammar rules dictated by the SIMPLE language. The diagram on the right represents possible happy paths that can be taken when validation is done using these helper functions. A point to note is that at each helper function, either another helper function is called on the same range of tokens or the range of tokens is restricted. This invariant prevents infinite loops from occurring.

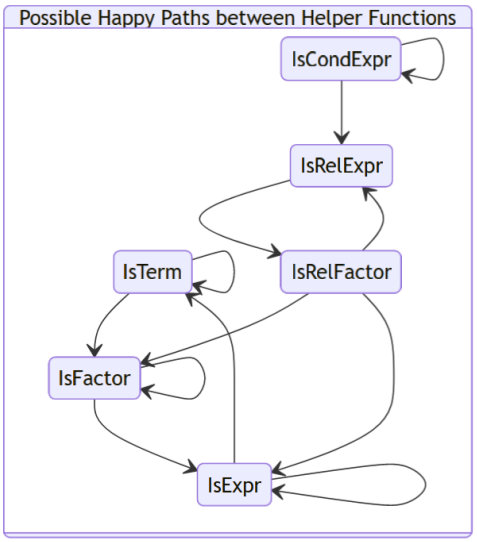


Figure 10 Happy Paths Possible for Validation Helpers

The Validation functions described above rely on multiple recursive calls to a set of helper functions described. We illustrate one such validation by running Statement #3 against the grammar rules:

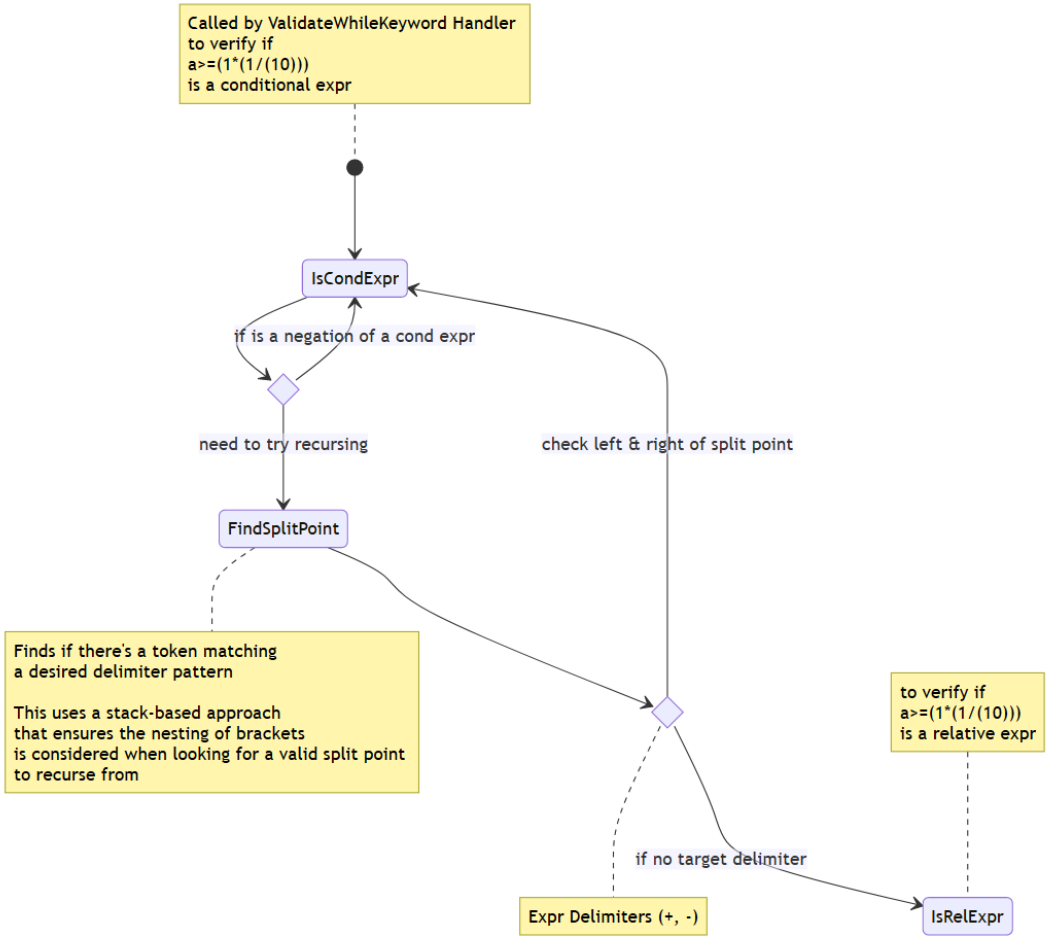


Figure 11 Logic in IsCondExpr Helper Function

The validation of `a>=(1*(1/(10)))` as a Relative Expression is then done as per the diagram on top.

The left-hand side of the relative expression, `a` is a valid factor hence it's a valid relative factor. For the right-hand side, `(1*(1/(10)))` is verified to be a valid Expression because it consists of valid terms.

SIMPLE statement #3 is a useful example to highlight the highly recursive nature of SyntaxValidation. As a summary, these are the actual calls made in validating that statement (diagram on the right):

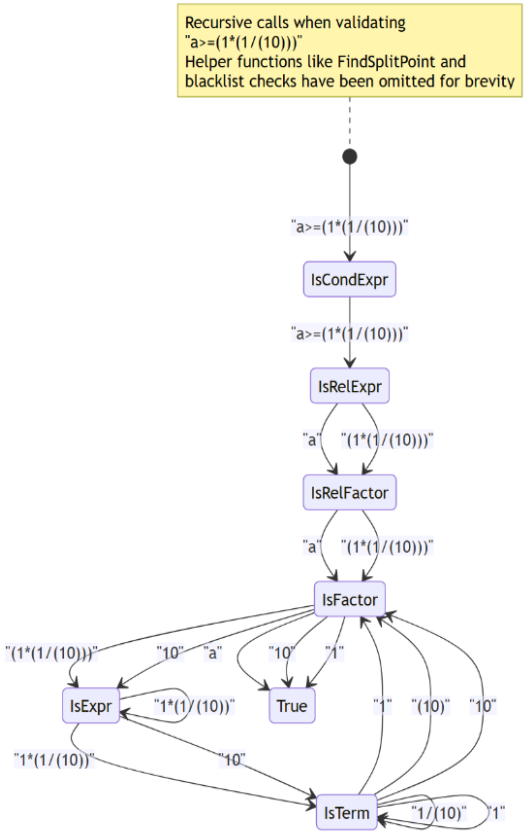


Figure 12 Recursive calls made when Validating "a>=(1(1/(10)))"*

Our SPA handles Semantic Validation as well. We define Semantic Errors to be the situation when the SIMPLE program is syntactically valid (i.e., does not violate the defined Concrete Grammar) but has some fallacies that affect the meaning of the program.

Broadly, Semantic Validation revolves verifying the meaning of the whole program, and hence we consider Semantic Validation to happen on an outer, more generic scope – on a Procedure and Program level.

This responsibility lies in the PSubsystem, which has access to such state and Syntax Validator, being responsible for validation on a per-statement basis, doesn't have the necessary access to such data. We postpone this description to after introducing the PSubsystem.

EntityFactory

The EntityFactory creates Entity objects from a vector of Tokens that make up a Statement. The delegation of this role to a separate class, that is the EntityFactory, allows the Parser and PSubsystem to work on the program level while the EntityFactory works on the Statement level. The following activity diagram shows the work done by the EntityFactory, using Sample Statement #1 as an example.

Steps 3 to 6 will differ according to the TokenTag matched. Upon the return of the method, PSubsystem will take appropriate actions on the Entity for further parsing.

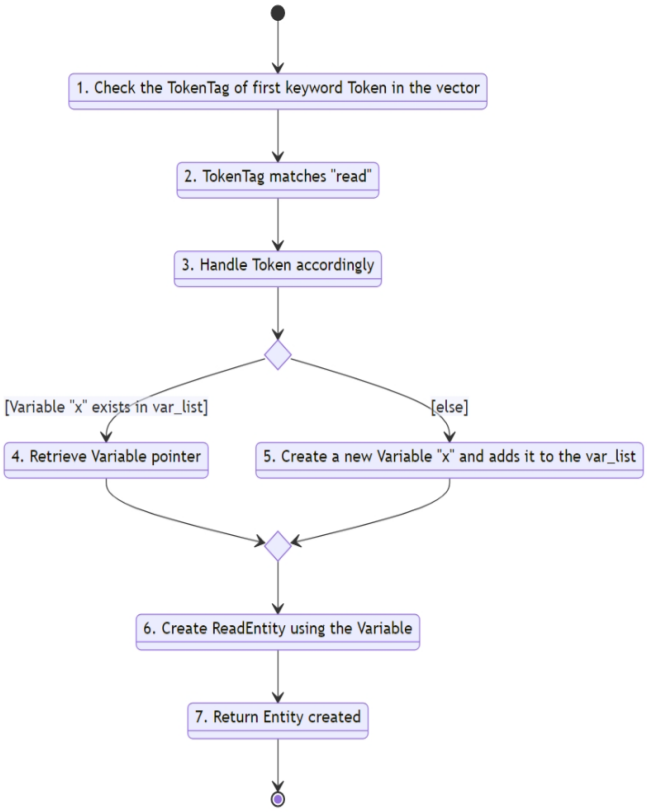


Figure 13 Activity diagram for creating entity for statement #1

1.3.2 Design extraction

Design extraction of SIMPLE source program is done in two places. Firstly, PSubsystem will update trivial relationship mapping (Parent, Follows, Uses and Modifies up to the container level, UsesP, ModifiesP, Calls), whose information are readily available, and DesignExtractor will perform non-trivial relationship mapping.

Deliverable

The Deliverable is a crucial construct in the Source Processor which collects the design extractions that will eventually be populated to the PKB. Design extractions are stored in simple primitive data structures, which are lists for Entities such as Procedure and AssignEntity, and hash-maps for relationships, that map the Entity to a list of its related Entities. The API for addition of design extractions can be distilled into 2 broad API calls below. While the API outlined here groups the specific Entities into its base class (Entity), the code implements specific API for each list and hash-map for the guarantee that the Entity added is of the correct type.

API	Description
Deliverable	
1. VOID AddEntity(ENTITY entity)	Adds the entity to its respective list of entities.
2. VOID AddRelationship(ENTITY entity, LIST<ENTITY> related_entities)	Adds the mapping of the entity to its related entities into the respective relationship hash-map.

Verifying Semantic Validation

The PSubsystem has the necessary access to program-level state for it to handle Semantic Validation of the Program. The Table below outlines some cases and how they are handled:

Case	Requirement	Validation Steps
1	Multiple procedures will not have the same name.	When encountering a new procedure, the Deliverable object's procedures list is read to ensure that this new procedure has a unique name.
2	Every call can only be done to a procedure that has been defined in the Program.	A finishing check is done by PSubsystem where the order of call statements and the order of Procedure declarations are compared and checked that all procedures are defined.
3	Recursive calls are not allowed.	This validation is done when extracting Transitive Relationships (call relationship) via a TransitiveExtractor which is next in the pipeline of components. By looking at the order of procedure calls, a potential cyclic procedure call can be detected.

Table 2 Semantic Validation Steps

PSubsystem

Below is a sequence of state diagrams running each of the first 5 lines as stated in Section 1.1. PSubsystem will update every object's before and parent object, as well as the relationship of Follows, Parent, Uses, and Modifies (first level), UsesP, ModifiesP and Calls into Deliverables.

The following 2 definitions refers to the members of a Statement, which will be relevant in the context of this scenario:

1. parent: Parent of this Statement
2. before: A Statement that comes before this Statement if any

The updating of the parent and before in each Statement object follows this logic:

1. parent: top object of parent stack
2. before: top object of follow stack if this Statement is not the first in its StmtLst

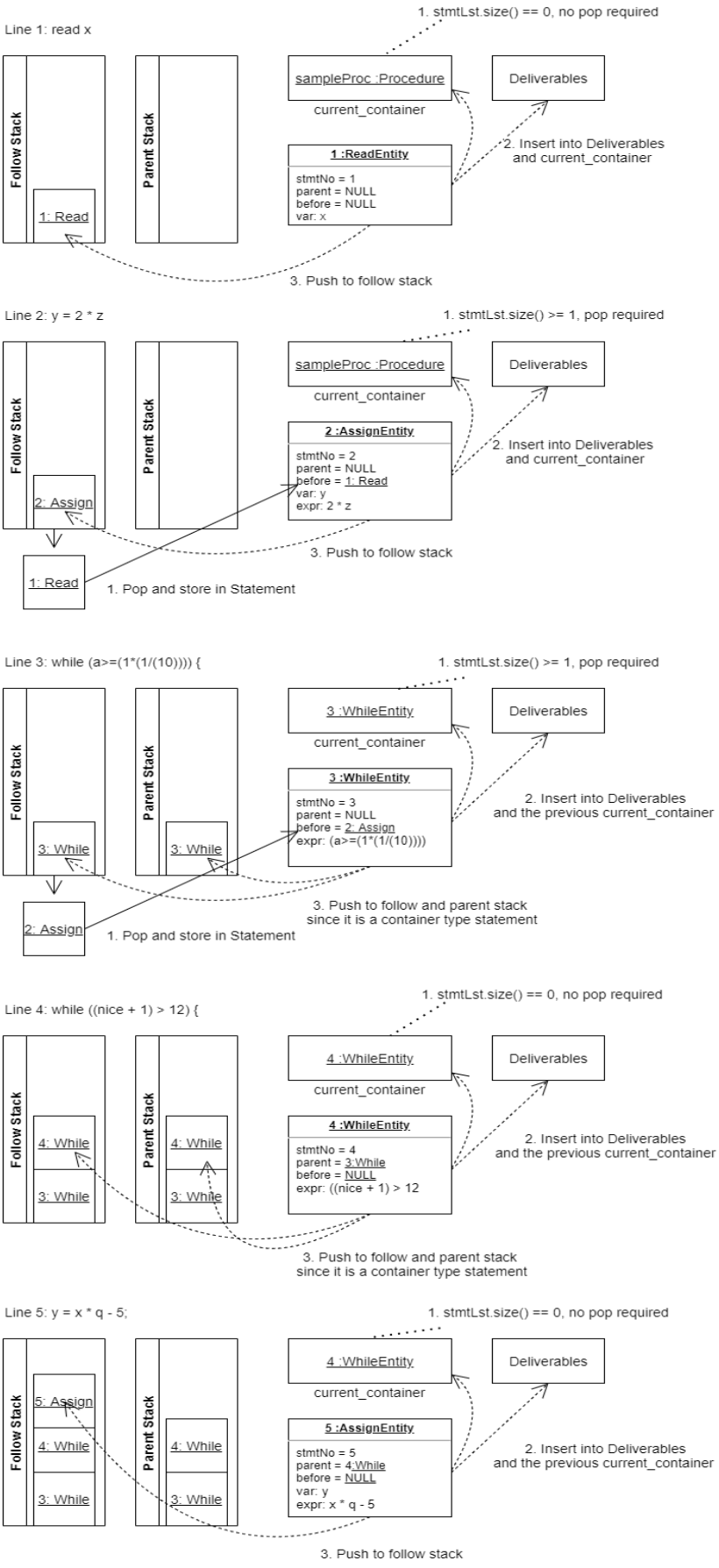


Figure 14 PSubsystem State Diagram

Upon reading a close brace "}", follow and parent stacks will pop the top of their stacks, current_container will update to current_container's parent variable.

After the follow and parent stacks are updated, for the relevant types of Statements, PSubsystem will add Uses and Modifies relationships between the Statement and the Variables in the Statement, as well as the relationship between the Container containing the Statement and the Variables.

In addition, with any update of Uses and Modifies for Statements, corresponding relationships for Procedures are also added, since all Variables in the Procedure are Used or Modified by the Procedure.

As for Calls, PSubsystem keeps track of the current Procedure so the Calls relationship between the current Procedure and the called Procedure can be added when it parses a Call Statement.

As such, trivial, non-transitive relationships will be added to the Deliverable in the first pass of parsing.

Representing Control Flow

The creation of the Control Flow Graph (CFG) is also performed on the same pass. In addition to that, our team also created another graph – Cluster Flow Graph (ClusterFG). ClusterFG is an extension to the normal CFG and has some nuanced implementation details that optimize path traversal. We refer to the nodes in CFG as `blocks` and nodes in ClusterFG as `clusters`. The figure on the right describes the classes involved.

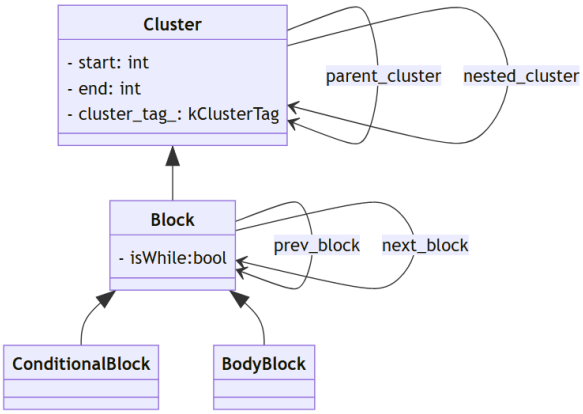


Figure 15 Class Diagram for Blocks & Clusters

The creation of Blocks and Clusters uses a stack-based approach that mirrors the previous state diagram. For brevity, we have provided this state diagram in the Appendix G

For blocks, our team closely followed the design description from class, adding an extra reverse pointer for every node in the CFG to support traversing through the CFG in reverse.

The motivation for each Cluster came from the way we as humans intuitively read through source code. We focus not on cyclical paths, but on how code blocks are nested within each other. Clusters are therefore a representation of acyclical control flow. This helps us use Clusters to skip regions of code for functionality that is reliant on the original CFG.

As an example, the example source code would have a cluster representation of these 3 clusters shown in Figure 16. Representation of blocks has been omitted for brevity and similarity to that suggested by the module.

The figure on the right shows how blocks and clusters are created within `Psubsystem`. We elaborate on the handling of Procedures, Container statements like `while` and `read` statement which is a non-container (no nesting).

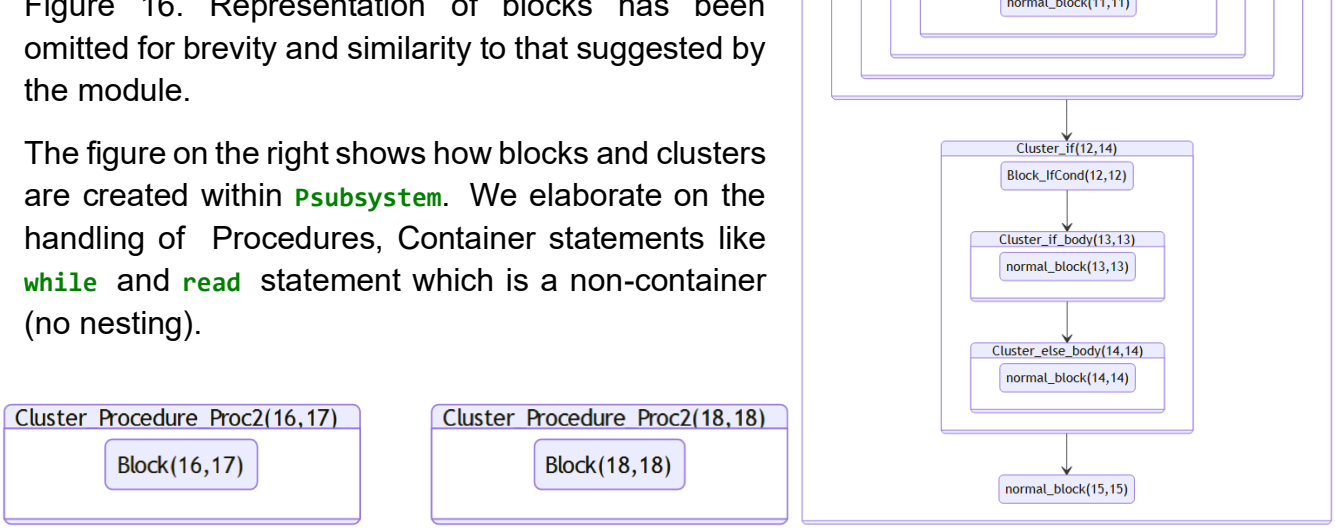


Figure 16 Creation of Blocks and Clusters in Psubsystem

The key implementation difference is that block stack will immediately discard the previous block when it is not required anymore, whereas for clusters, it will be discarded whenever a close brace "`}`" is encountered.

Clusters can be recursively traversed since `while-clusters` will have 2 nested clusters (`while-cond` and `while-body cluster`) and `if-Clusters` will have 3 nested Clusters (`if-cond`, `if-body cluster` and `else-body cluster`).

DesignExtractor

The DesignExtractor acts as a controller and is responsible for extracting transitive relationships from non-transitive relationships stored in the deliverable. The implementation of extracting the different relationships is abstracted by the Extractor subclasses, namely TransitiveExtractor, VariableTExtractor and NextTExtractor.

The sequence diagram below shows the interactions between the DesignExtractor and the Extractor subclasses for extracting relationships.

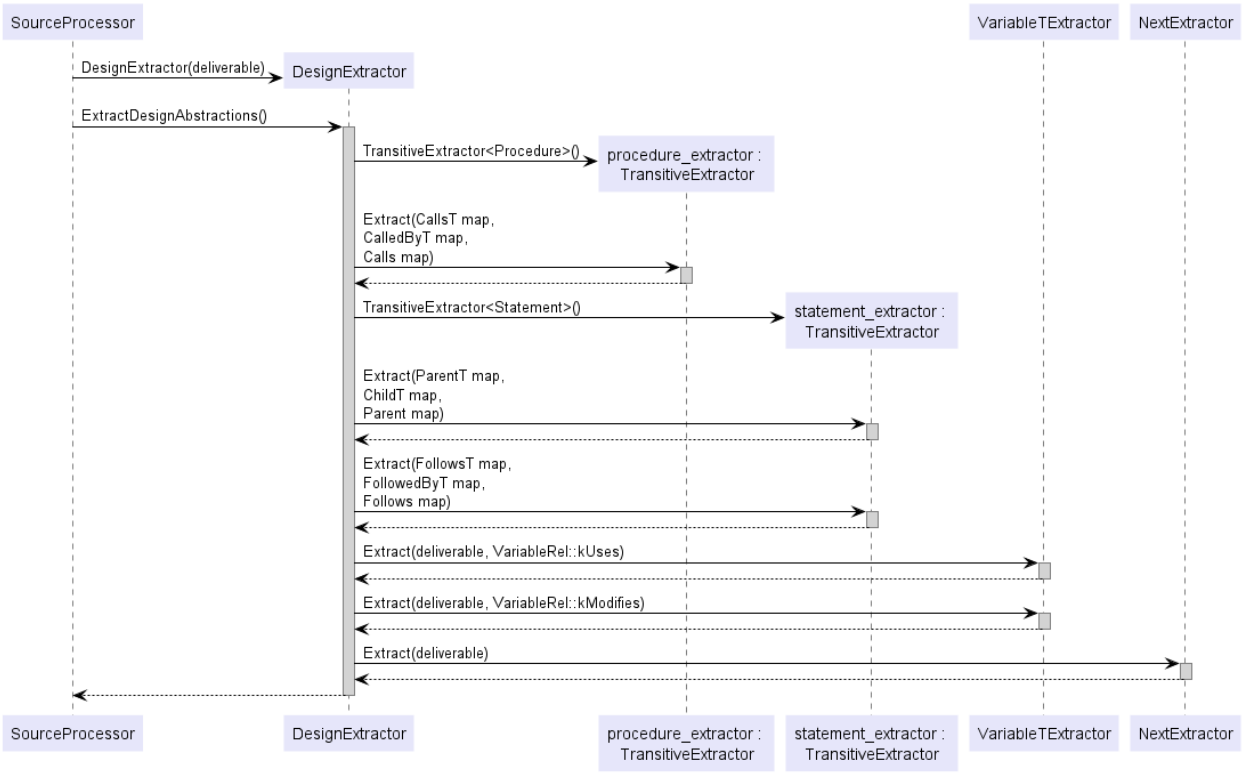


Figure 18 sequence diagram for DesignExtractor

Responsibilities of each Extractor:

- TransitiveExtractor extracts transitive relationships from non-transitive relationships. (Calls*/Follows*/Parent*)
- VariableTExtractor extracts transitive relationships between Container statements and variables. (Uses/Modifies)
- NextExtractor extracts Next relationships from the CFG.

All these Extractors add to the relevant relationship hash maps in the Deliverable while extracting.

The algorithms used in the Extractor subclasses have been implemented with the concept of DFS in mind, while adapting it to the requirements of our system. The following activity diagrams will illustrate the steps taken by each Extractor in extracting their respective relationships.

TransitiveExtractor

The TransitiveExtractor applies DFS to non-transitive relationships to extract the transitive relationship and adds them to the transitive relationship hash-map reference passed into the extraction method. Since it's a template class, it can be reused for extracting Calls*, Follows* and Parent* relationships. An example of extracting Calls* is elaborated below.

The algorithm used here applies DFS with Procedures as nodes and Calls relationships as edges. While adding a Calls* relationship at step 8, if the nodes are the same, then a cyclic relationship is detected and will throw an exception immediately.

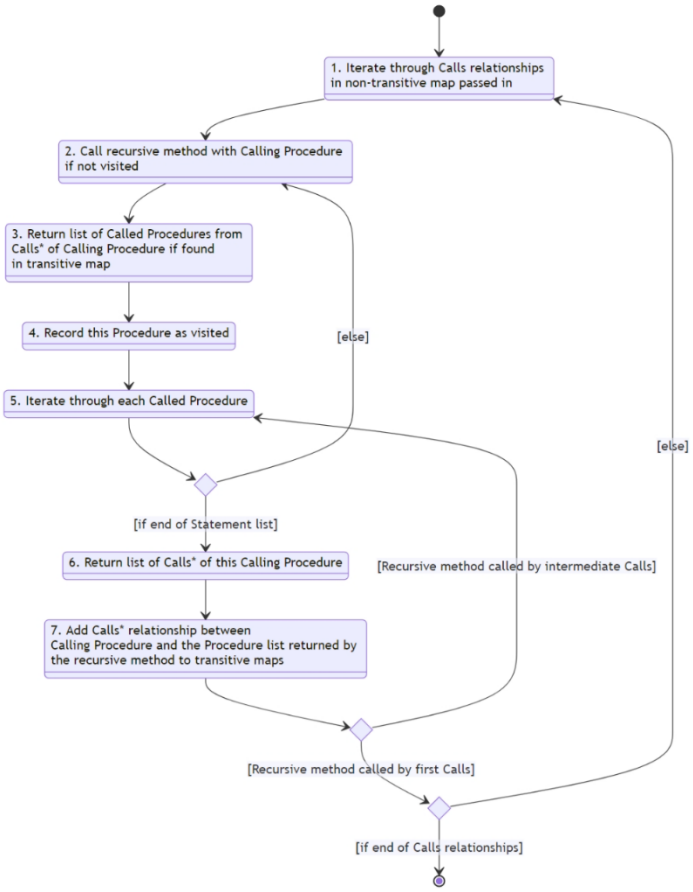


Figure 19 Calls Transitive Extractor

Using the sample code from Section 1.2, the data extraction sequence is as follows:

1. Step 2 is called by the first Call in the first procedure, which is statement #15
2. Step 3 is skipped as it is the first visit of this Call statement
3. Step 2 is called by the Procedure in this Call statement and steps 3 to 5 are repeated for Proc2
4. Then steps 6 and 7 occurs for the Call statement in the order of statements #16, then #15.
5. At step 7, if any Calls relationship is added between the same procedures, an exception will be thrown indicating a cyclic and/or recursive call.

VariableTExtractor

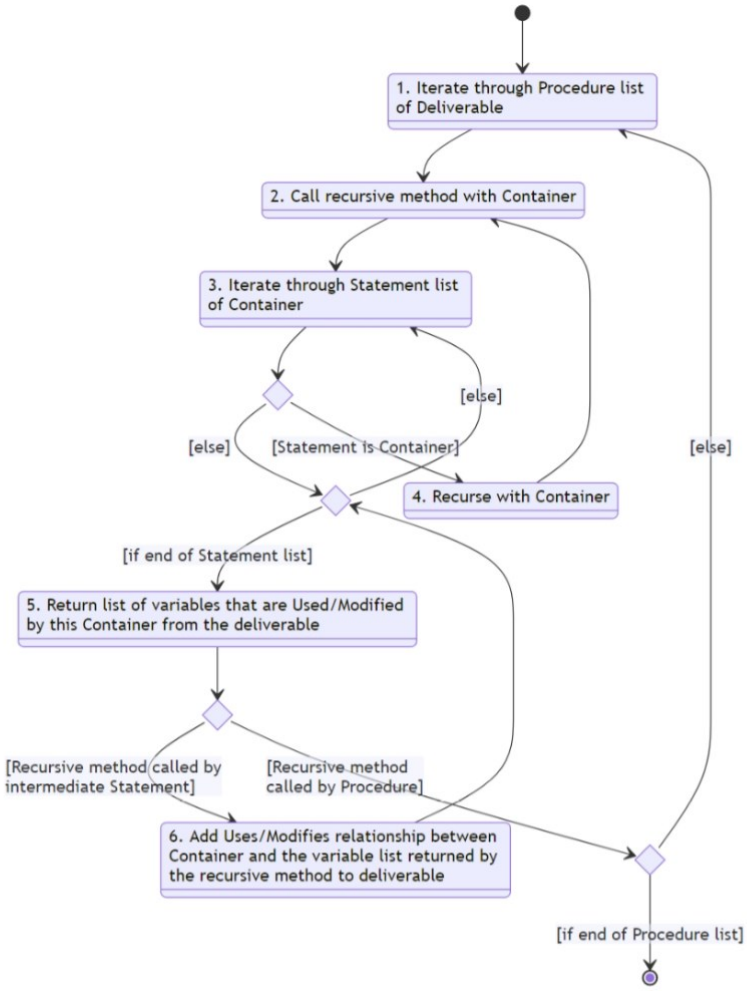


Figure 20 Activity diagram for extracting Uses/Modifies

In the application of DFS to this use case, Container statements are treated as vertices and the container-contained relationship as edges. The difference between DFS and this adapted algorithm is that IfEntities and WhileEntities are treated differently due to the composition of ElseEntities in IfEntities. This behavior happens between steps 3 and 4 and is omitted from the diagram for brevity. Then in step 6, after the call to the recursive method, the current relationships accumulated will be added to the hash-maps in the deliverable.

Using the sample code from Section 1.2, the data extraction sequence in the first while loop in statement #3 is as follows:

1. Step 2 is called by the first Procedure
2. Step 4 is called by statement #3, 4, 7 in order
3. Step 5 and 6 occurs in the order of last called, which is statement #7, 4, 3
4. Then step 3 continues with the rest of the program after the closing of the while loop in statement #3

NextExtractor

The NextExtractor extracts Next relationships in the DesignExtractor after the CFG has been created. It relies on the links between blocks.

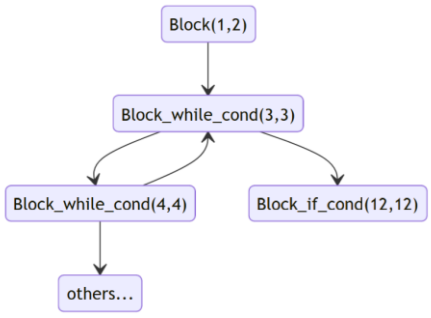


Figure 21 Partial CFG of the SampleProc procedure

The steps to extract from the CFG for the SampleProc procedure is as follows:

1. Add Next relationship to the hashmap in the Deliverable for every consecutive statement in a block.
2. Add Next relationship for an outgoing arrow of a block, if any.
3. For blocks with multiple outgoing arrows such as `Block_while_cond(3,3)`, extract completely for one path before extracting the other path.
4. Visited blocks will be marked and not traversed again.

1.3.3 Populating the PKB

In the Source Processor component, relationships and entity tables are first recorded in the Deliverable wrapper object, as previously mentioned in high level data flow sequence diagram of 1.3 Source Processor and in 1.3.2 Design Extraction. Acting as a Data Transfer Object, the Deliverable is passed along the pipeline described in 1.3 Source Processor and finally populates the PKB with the relevant method call `PopulateDataStructures` in one go. This populate method is further explained in the section for PKB.

1.3.4 Design decisions

During the design of Source Processor, we encountered several design decisions to be made. In the following table, our team will discuss about our pick of using such designs as well as the deciding factors of our choices.

Legend: Advantages [✓] | Other factors [▪] | Disadvantages [–]			
Design decisions		Team's pick	Deciding factors
1	Reading from source file in 1 pass	Our team decided to utilize the single pass reading to be efficient.	✓ Efficiency ✓ Lesser time usage overall with slightly higher memory to store stacks

		Stemming from that, PSubsystem contains the immediate parent and before stack to perform backtracking. With the readily available data, 1-level relationship is also extracted in the pass.	✓ Immediate relationship is extracted ▪ Transitive relationships are ignored to prevent time wastage on unnecessary popping and pushing of stack - May be harder to test due to complexity of the code
2	Each Statement object contains related and neighboring objects.	For every Statement, it will contain pointers to the parent and before objects. For every Container, it will store a list of Statement objects.	✓ Gives as much information to Design Extractor and PKB to allow greater flexibility in subsequent extractions. ✓ Allows easier access to related statement objects. ✓ Gives easy access to the entire AST network from any Statement object within the AST. - Proper memory management and disposal of unnecessary information after parsing is required.
3	Pipeline Structure	Our team decided to evaluate Statements in a pipeline manner. At each stage of the pipeline, there is a clear transformation of data.	✓ Clear input and output to aid in unit testing ✓ Ease of extension as functions can be easily inserted or removed
4	Encapsulation of design abstractions in a DTO to populate PKB	Design abstractions are encapsulated in a Deliverable object, which functions as a Data Transfer Object (DTO). This Deliverable is passed down the Source Processor pipeline and finally populates the PKB. This design choice is chosen over accessing	✓ Single “write” operation reduces communication with PKB ✓ Subcomponents in Source Processor work on the Deliverable instead of PKB, reducing coupling across major components ✓ Extending the Deliverable to include more data structures is not any more difficult than when accessing the PKB via setter calls

		the PKB via setter calls.	- A change in data structures in the Deliverable will require a change in the populating method of PKB, though we are not expecting many modifications in the specifications

Table 3 Source Processor Design Decisions and Considerations

For assign statements, our team had considered a few deciding factors to choose which data structure we used to store the expression. We have concluded by comparing the following criteria: Speed of checking, Ease of implementation and Space Requirement.

Criteria	Postfix Expression	AST (as given in class)
Speed of Checking	To match a subset to a postfix, it must match a substring of the original postfix expression. The longest it can take is when the entire postfix does not contain the substring, being O(n).	The match a subset to an AST, it must match the root operator, followed by its child nodes. This is like matching a subtree within an AST. The worst-case scenario being it does not match any subtree, being O(n).
Ease of Implementation	From the given infix to postfix notation, using of a stack is sufficient. It is slightly easier to implement as postfix handles it iteratively.	From the given infix to AST, one must be careful of the insertion order of the AST due to the operator precedence. It is also slightly harder to do as it is recursive in nature.
Space Requirement	The space required to store the expression is dependent on the assign statement. It has significantly lesser overheads as 1 character in assign statement would translate to 1 character in string.	In addition to the space required to store each operator and IDENT, additional space is required to store the pointers for traversal. This can include forward and backward pointers, or any auxiliary pointers to assist in searching.

Table 4 Assign Statement Design Decisions

With the above design considerations, our team had decided to use postfix expression to store assign statements due to its relative ease of implementation and lesser space required.

Ultimately, while the alternatives to the above design decisions are valid, we were able to come up with a cleaner architecture of our system that can meet the SPA requirements well. The class diagram below shows our completed class structure.

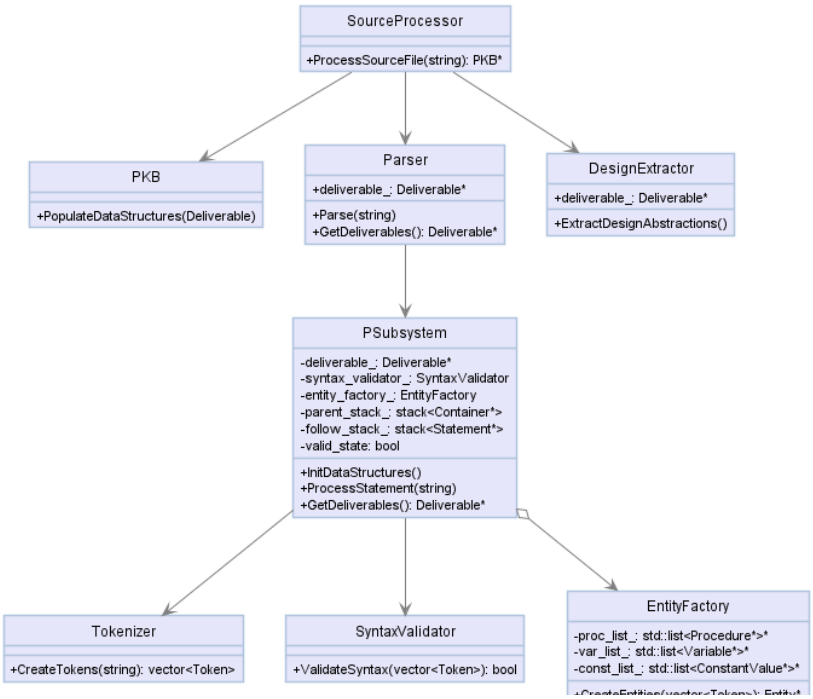


Figure 22 Source Processor class diagram

1.3.5 Abstract API

API	Description
SourceProcessor	
1. PKB* ProcessSourceFile(STRING sourceFilePath)	Processes the source file at the path given, and returns a PKB pointer to the PKB that is populated with the necessary data.
Parser	
2. VOID Parse(STRING sourceFilePath)	Parses the source file at the path given, by calling the PSubsystem.
PSubsystem	
3. VOID ProcessStatement(STRING stmt)	Tokenizes the statement, validates its concrete syntax and creates Entities and relationships that will then be added to the Deliverable.
Tokenizer	
4. TOKEN_LIST CreateTokens(STRING stmt)	Creates tokens out of the statement according to the Syntax.
SyntaxValidator	
5. BOOL ValidateSyntax (TOKEN_LIST tokens)	Validates the syntax of the vector of tokens.
EntityFactory	
6. ENTITY* CreateEntities (TOKEN_LIST tokens)	Creates Entities based on the statement represented by the vector of tokens passed in.
DesignExtractor	

7. VOID ExtractDesignAbstractions( )	Extracts program design abstractions from the data available after parsing and populates the relevant tables.

Table 5 Source Processor Abstract API

1.4 DBManager

1.4.1 Design decisions

There is a distinction to be made between relationships that exist within the PKB with those that do not. Namely, the Next*, Affects and Affects* relationships must be computed during runtime rather than preprocessing. Therefore, computing these relationships is done separately in the RuntimeExtractor. Otherwise, all relationships that can be preprocessed are in the PKB.

The QueryEvaluator therefore may not know whether to call the PKB or the RuntimeExtractor for specific relationships. The DBManager therefore redirects the query to either the PKB or the RuntimeExtractor. In this way, the DBManager adopts the Proxy structural design pattern. In this way, we can abstract out this logic and responsibility from the QueryEvaluator. The public API method signatures of the DBManager therefore match the PKB and the RuntimeExtractor exactly.

1.5 Program Knowledge Base (PKB)

1.5.1 Design decisions

The PKB is designed to optimize for the speed and efficiency of query evaluation. To optimize query evaluation, the PKB is structured in such a way that the fewest number of API calls from the QueryEvaluator is required for each query. The design choice of the PKB is therefore reliant on the QueryEvaluator, since we must first determine, what information is needed by the QueryEvaluator.

The population of the PKB is done through a Deliverable object, by means of the `PKB::PopulateDataStructures` method. While the Deliverable object contains all the information that will be stored in the PKB, the PKB arranges this information considering the considerations laid out in the previous paragraph. The population of the PKB occurs completely during preprocessing.

The bulk of the data within the PKB is stored within four data structures, each of which will be discussed in more detail. As far as possible, the data structures within the PKB that carry similar information have been combined. This helps to remove duplicate code and therefore lessen the likelihood of bugs. When choosing our data-structures, the guiding principle for our choices were that wherever possible, we use hash maps for relationship mappings and use vectors (for example, for example as compared to list) for faster direct access since memory is contiguously allocated when using vectors. Namely, these data structures are for type-entity mapping, relationship-mapping by entity names, relationship mapping by entity types,

relationship set, pattern mapping and attribute mapping. Other auxiliary data structures have been created as well to assist with internal logic.

The logic for pattern matching is found within the Entity objects themselves rather than in the PKB. This means that each Entity object (representing an Assign, While or If statement) has methods to check whether it matches a specific pattern. However, for quick access, these Entity objects can be retrieved by their variables. For instance, given the fact that I want to find all entities of while-type that have “x” as a control variable, I can do this in O(1) time.

1. **Type-Entity Mapping**

This data structure consists of a hash map mapping from entity type to a vector of entities of that type. This structure is meant to allow retrieval of all entities of a specified type in O(1) time.

2. **Relationship-Mapping by Entity Names**

This data structure is a multi-tiered hash map. The key to the outermost hash map is the relationship type (Follows, Parent, Uses etc.). The value is another hash map mapping from entity names to a vector of all entities that possess the specified relationship with the entity of the specified name.

Given a such-that clause, for instance “such that Follows(1, s1)”, all possible values of s1 can be retrieved in O(1) time.

3. **Relationship-Mapping by Entity Types**

This data structure is created to abstract some logic away from the query evaluation, through preprocessing what relationships exist between which types. Once again, this data structure is a multi-tiered hash map, with the key to the outermost hash map being the relationship type. The value is another hash map mapping from all possible pairs of entity type to a vector of pairs of entities. Each pair of entities are of the specified entity types and the specified relationship holds between them.

Given a such-that clause, for instance “such that Uses(s1, v1)”, the Query Evaluator can therefore retrieve all possible pairs of values of s1 and v1 respectively. There are also separate hash maps of similar structure which store vectors of only the first or second entity.

4. **Relationship Set**

This data structure serves as a store for all relationships that exist within the given input SIMPLE program. It is a set of all relationship, consisting of the relationship type as well as the two entities that possess the relationship with one another. The set is meant to allow the Query Evaluator to check if a given relationship exists in O(1) time.

Given a such-that clause, for instance “such that Follows(3, 4)”, the Query Evaluator can therefore evaluate this Boolean clause with a single call to the PKB.

5. **Pattern Map**

This data structure was created to allow lookup of pattern-matching entities by their variables. For While and If statements, these refer to all control variables. For Assign statements, this refers to the variable being assigned. The data structure consists of a multi-tiered hash map, in which the key to the outermost hash map is the entity type. The value is another hash map whose key is the variable name and whose value is a vector of statements. Given “pattern w (“a”, _)”, we can therefore retrieve all matching While entities in O(1) time.

6. **Attribute Maps**

There are two main data structures that store attribute data. These are needed for evaluation of ‘with’ clauses. The first data structure is a hash map of entity type, attribute type and attribute value to a vector of entities. This allows quick lookup of entities with specified attribute values. The next data structure is a mapping of two entity/attribute type combinations to a vector of entity pairs. This allows for lookup of pairs of entities which have matching attribute values. For example, given the clause “with c1.procName = v1.varName”, we can retrieve in O(1) time all possible values of c1 and v1.

1.5.2 Alternative Designs

We considered various implementations for our PKB, and iteratively improved upon each one. Initially, our PKB was to consist solely of an Abstract Syntax Tree (AST). Edges of different types would be created between any nodes with relationships to one another. Any queries would then be evaluated during runtime. For example, given “Follows(1,2)”, the AST would first need to be traversed till statements 1 and 2 were found, and then be checked for a “Follows” edge between them. We opted against this implementation due to the high latency during runtime (from tree traversal). Our next implementation involved storage of all relationships (transitive and non-transitive) as well as entities within hash maps in the PKB. Relationship and entity retrieval could then be cut down to O(1) time given an entity’s name or type, respectively. To allow for fast evaluation of queries, more hash maps were subsequently created to allow retrieval of entities by different criteria, as outlined above in 1.5.1 Design Decisions.

1.5.3 Abstract API

An abridged version of the PKB abstract API is below. In addition to the following, the PKB also has getter functions for Assign, While and If statements by their statement numbers and the variables within them.

API	Description
PKB	

1. VOID PopulateDataStructures (DELIVERABLE d)	Receives a Deliverable object, containing all entities and relationships between those entities, and stores that information within curated data structures within the PKB.
2. RELATIONSHIP_MAP GetRelationshipMap (REL_TYPE rs_type)	Receives a relationship type and returns a map of entity names to other entities in a relationship with the specified entity.
3. ENTITY_LIST GetRelationship (REL_TYPE rs_type, ENTITY_NAME entity)	Receives a relationship type and an entity name (Statement number, variable name, procedure name) and returns a list of all entities that possess the relationship with the specified entity. Returns an empty list if there is none.
4. ENTITY_LIST GetRelationshipByType (REL_TYPE rs_type, ENTITY_TYPE entity_type)	Receives a relationship type and an entity type and returns a list of entities of the specified entity type that possess that relationship. Returns an empty list if there is none.
5. ENTITY_PAIR_LIST GetRelationshipByTypes (REL_TYPE rs_type, ENTITY_TYPE type_1, ENTITY_TYPE type_2)	Receives a relationship type and two entity types and returns all pairs of entities of the specified types between which the specified relationship holds. Returns an empty list if there is none.
6. ENTITY_LIST GetDesignEntities (ENTITY_TYPE entity_type)	Receives an entity type and returns a list of all entities belonging to that type. Returns an empty list if there is none.
7. ENTITY_LIST GetPatternEntities(ENTITY_TYPE entity_type, ENTITY_NAME var_or_stmt)	Receives an entity type (While, Assign or If) and a variable name or statement number and returns matching pattern entities of the specified type.
8. ENTITY_LIST GetEntitiesWithAttributeValue(ENTITY_TYPE entity_type, ATTRIBUTE_TYPE attribute_type, ATTRIBUTE_VALUE attribute_value)	Receives an entity type, attribute type and attribute value and returns a vector of entities of that specified type with that attribute value.
9. ENTITY_PAIR_LIST GetEntitiesWithMatchingAttributes(ENTITY_AND_ATTRIBUTE_TYPE	Receives two Entity/Attribute type combinations and returns a list of all entity pairs of those types that have matching attribute values.

type_1, ENTITY_AND_ATTRIBUTE_TYPE type_2)	
10. BOOL HasRelationship(REL_TYPE rs_type)	Receives a relationship type and returns a Boolean of whether the relationship exists in the source SIMPLE code or not.
11. BOOL HasRelationship(REL_TYPE rs_type, ENTITY_NAME entity)	Receives a relationship type and an entity name and returns a Boolean of whether the specified entity has the relationship.
12. BOOL HasRelationship(REL_TYPE rs_type, ENTITY_TYPE type_1, ENTITY_TYPE type_2)	Receives a relationship type and two entity types and returns a Boolean of whether the relationship exists between any two entities of the specified types in the source SIMPLE code or not.
13. BOOL HasRelationship(REL_TYPE rs_type, ENTITY_NAME entity_1, ENTITY_NAME entity_2)	Receives a relationship type and two strings identifying entities and returns a Boolean of whether a relationship of the specified type between two entities with those identifiers exists in the source SIMPLE code or not.

Table 6 PKB Abstract API

1.6 Runtime Extraction

Runtime extraction encompasses the extraction of runtime relationships. A runtime relationship is defined as being Next*, Affects, Affects* and other Bip relationships. When a query to the `DBManager` is made for a runtime relationship, the query will be proxied to the `RuntimeExtractor`, which implements the mediator design pattern.

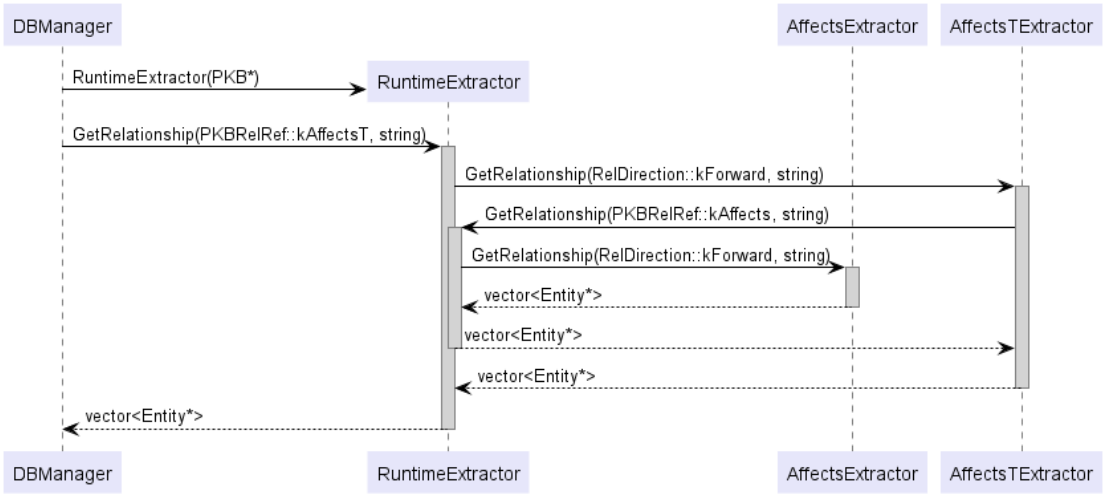


Figure 23 RuntimeExtractor mediating AffectsExtractor and AffectsTExtractor

The RuntimeExtractor implements the RuntimeMediator interface and composes of the following classes:

- NextTExtractor
- AffectsExtractor
- AffectsTExtractor
- Bip extractors which are outlined in the Extensions section

Each extractor has a reference to the PKB to retrieve necessary information but stores relationships in its own object as hash-maps that map the Entity to its related Entities. They implement a common RuntimeColleague interface that make the RuntimeExtractor easily extendable. The API used for both the RuntimeColleague and RuntimeMediator is outlined in the Abstract API section. For optimization purposes, the relationships are extracted only in the context of a PQL clause and not the entire source program.

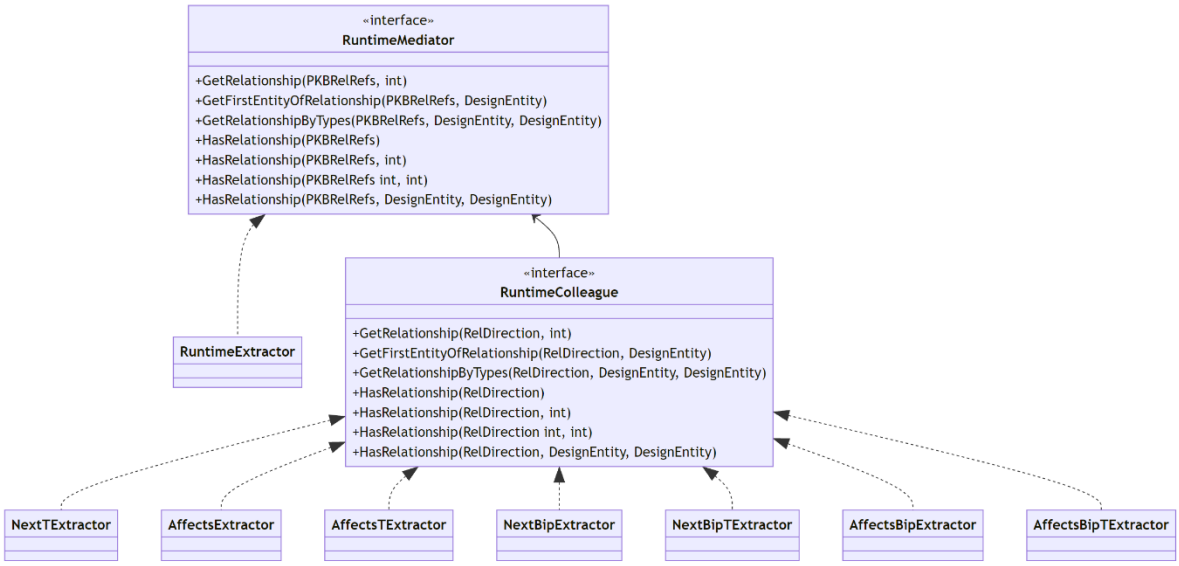
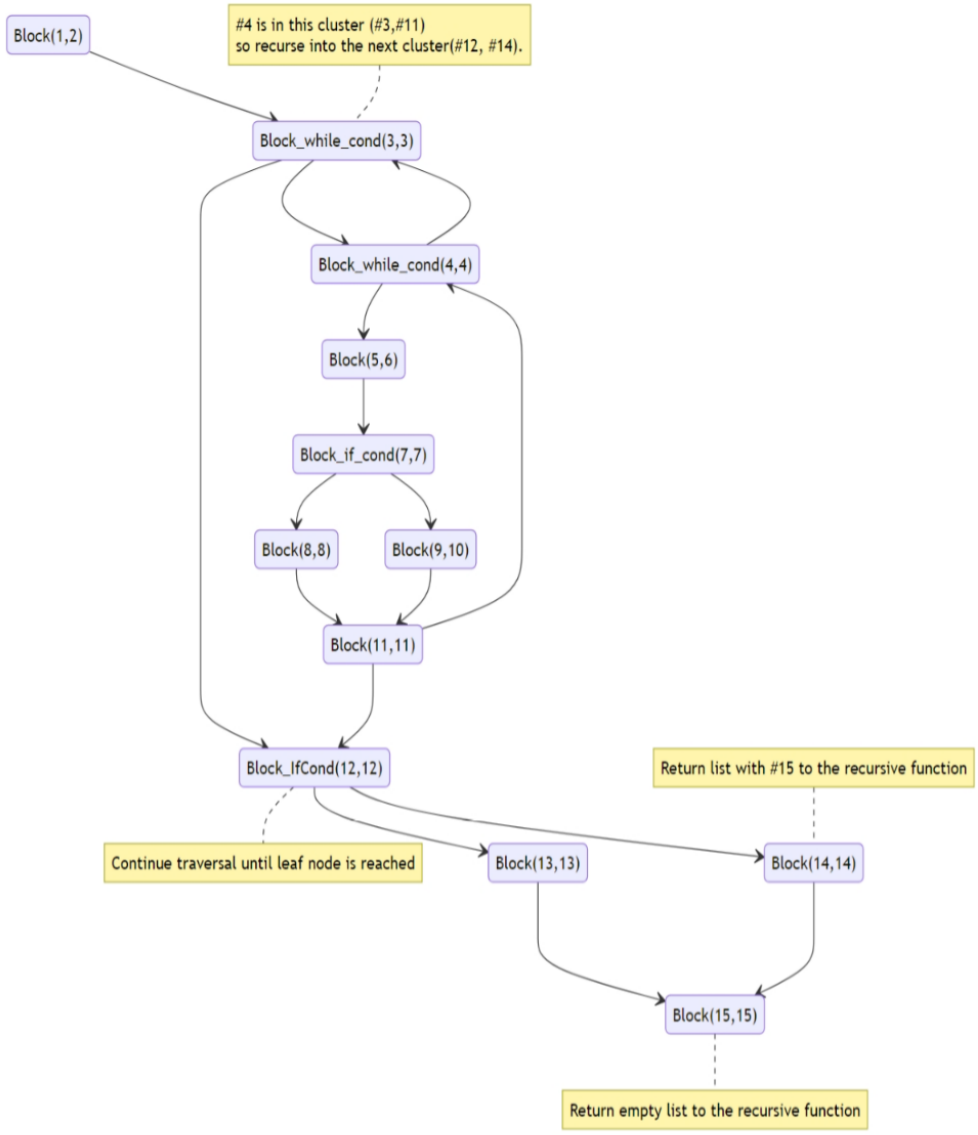


Figure 24 Runtime Mediator Pattern for Runtime Relationships

1.6.1 Design Extraction

NextTExtractor

In general, NextTExtractor implements CFG traversal for extracting Next* relationships for `GetRelationship(REL_TYPE rs_type, INT entity)` and `HasRelationship(REL_TYPE rs_type, INT entity_1, INT entity_2)` calls. All the other methods are derivations of these 2 methods.



This is a walkthrough of the CFG traversal when GetRelationship(kNext, 4) is called and starts from the top of the CFG.

When the recursive function returns to the While cluster at #3, for each statement in the while cluster (#3, #11) add Next* relationship to every statement.

For HasRelationship, the function can terminate much earlier when the second argument is found in the same cluster as the first argument or is downstream.

AffectsExtractor

For AffectsExtractor, all the different variations of HasRelationship and GetRelationship are derived from HasAffects(x,y) where x and y are statement numbers. This allows us to perform just-in-time evaluation as well as cache results whenever they are returned. All functions utilize the HasAffects function, hence the core logic of Affects extraction is kept to a single point, making bug-fixing more manageable.

In the following activity diagram, we show how to evaluate the HasAffects relationship.

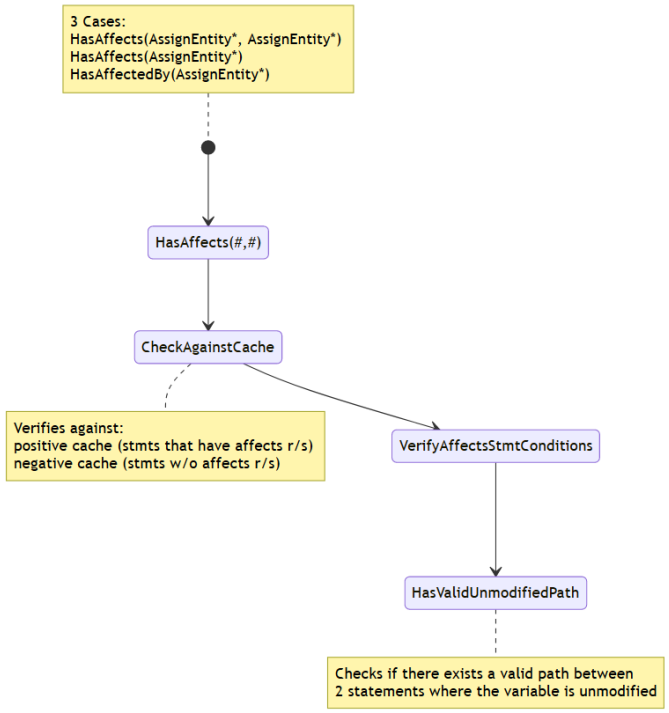


Figure 25 Control Flow for HasAffects(#,#)

`TraverseScopedClusterForAffects` is the core function that takes in the innermost cluster than contains both the first and second statements (the `ScopedCluster`). A series of steps eventually determine if there exists a valid path between the two statements by traversing using the Cluster as outlined below.

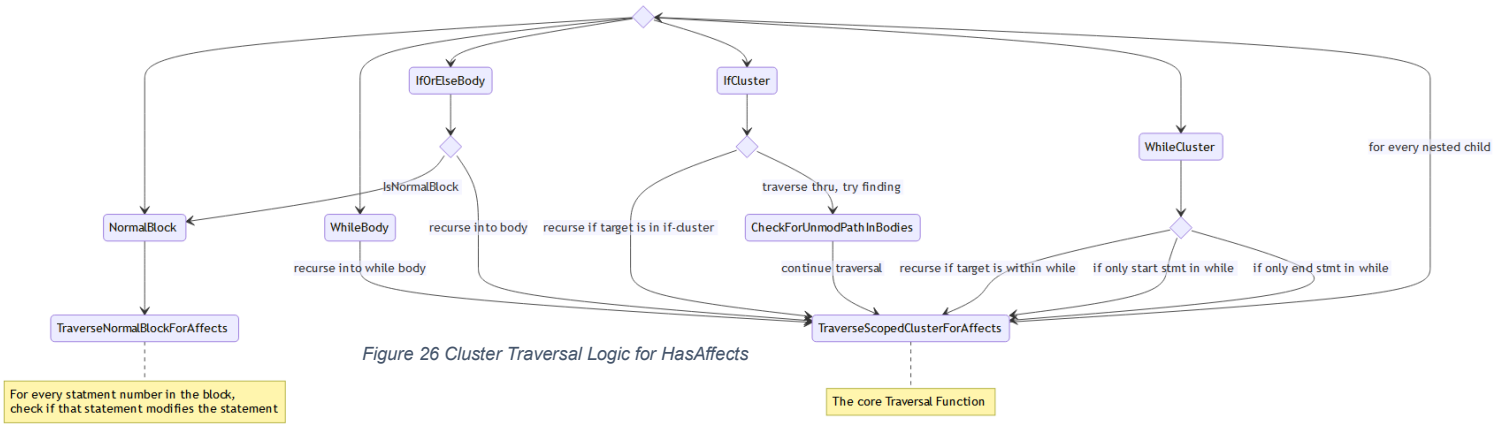


Figure 26 Cluster Traversal Logic for HasAffects

AffectsTExtractor

For AffectsTExtractor, our extractor will call AffectsExtractor's GetRelationship to get all affects relationship and store it into cache. For subsequent evaluation of AffectsT, it will use this cache, and if required, perform a Breath-First Search to retrieve any transitive relationships.

1.6.2 Design Considerations

The RuntimeExtractor was created with a few considerations in mind and was intentionally kept separate from the DesignExtractor, as well as caching its own relationships. The reasons are laid out below.

Criteria	**Extract runtime relationships in the Design Extractor**	**Extract in a separate Runtime Extractor**
Coupling between components	Increases coupling across major components.	Coupling is confined in the RuntimeExtractor and its extractor classes and can apply the Mediator pattern to manage the interactions between those classes.
Defining responsibilities of components	Violates SRP when the Design Extractor is extracting from different data constructs from the Deliverable and the PKB.	Ties in with the decision to use a Data Transfer Object to populate the PKB in one go and keep the PKB data structures unmodifiable once populated.

Table 7 Design consideration of creating a new component for extracting runtime relationships

Criteria	**Caching relationships in PKB**	**Caching relationships in runtime extractors**
Cache management	Must selectively delete the cached relationships after every query.	Being objects that are created with the DBManager, the cache is automatically garbage collected after the query.
Cohesion of classes	Runtime extractors will require more auxiliary data to manage just-in-time extraction. This makes the class dependent on both the PKB and its own auxiliary data members, resulting in lower cohesion.	The data cache in each extractor will only deal with its respective relationship, increasing its cohesion and maintainability.

Table 8 Design consideration of caching relationships in runtime extractors

1.6.3 Abstract API

The RuntimeMediator and RuntimeColleague use the same signatures for subset of methods from the DBManager and PKB, namely:

S/N	API Method
1	ENTITY_LIST GetRelationship(REL_TYPE rs_type, INT entity)
2	ENTITY_LIST GetFirstEntityOfRelationship(REL_TYPE rs_type, ENTITY_TYPE entity_type)
3	REL_PAIR_LIST GetRelationshipByType(REL_TYPE rs_type, ENTITY_TYPE
4	BOOL HasRelationship(REL_TYPE rs_type)
5	BOOL HasRelationship(REL_TYPE rs_type, INT entity_1)
6	BOOL HasRelationship(REL_TYPE rs_type, ENTITY_TYPE entity_1_type, ENTITY_TYPE entity_2_type)
7	BOOL HasRelationship(REL_TYPE rs_type, INT entity_1, INT entity_2)

1.7 Query Processor Subsystem

The following sequence diagram gives an overview of the Query Processor.

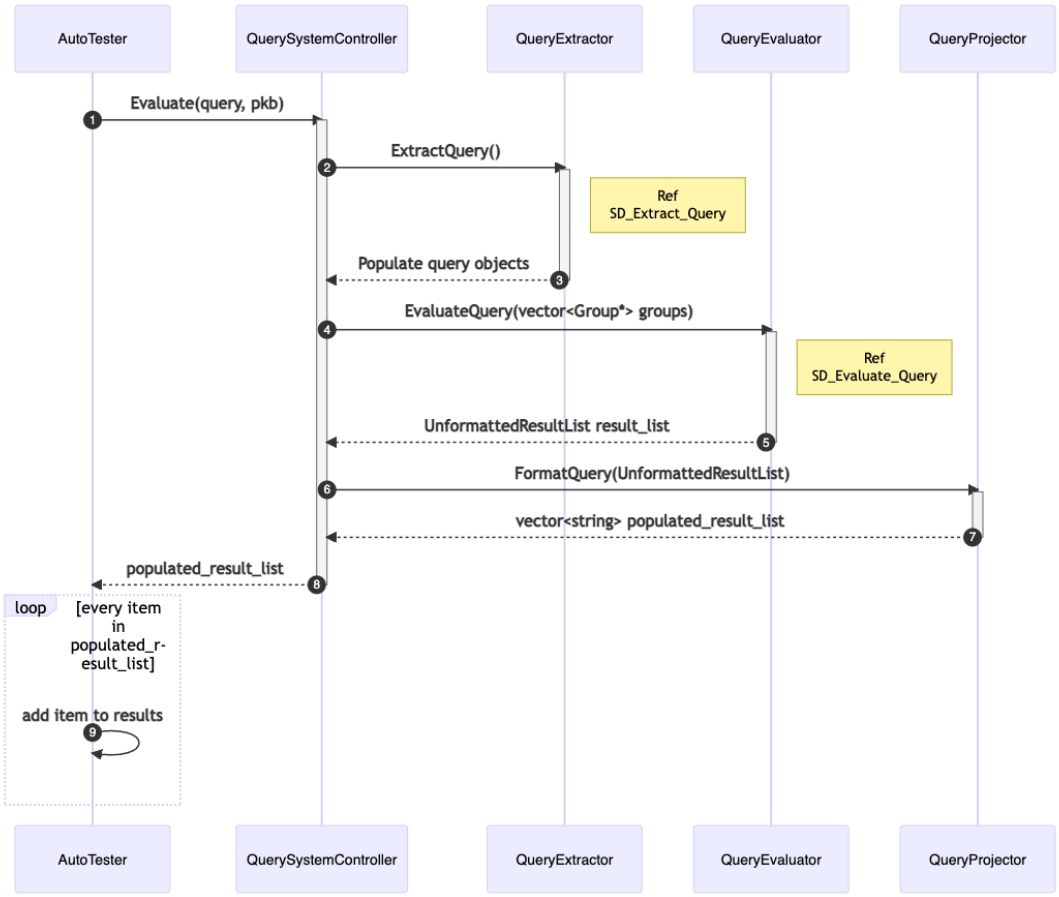


Figure 27 Query Processor sequence diagram

As shown in the diagram above, the overall communication between the AutoTester and our Query Processor happens solely via interaction with the QuerySystemController. As the name suggests, it uses the façade design pattern to hide implementation details from AutoTester. Logically, the QuerySystemController calls the aptly named subcomponents to extract information from a query input stream, to then evaluate the query with the extracted information, and finally to suitably format the results for projection back to the caller.

We provide information on the overall extraction and evaluation of the query in the two subsections that follow.

1.7.1 Query Extraction

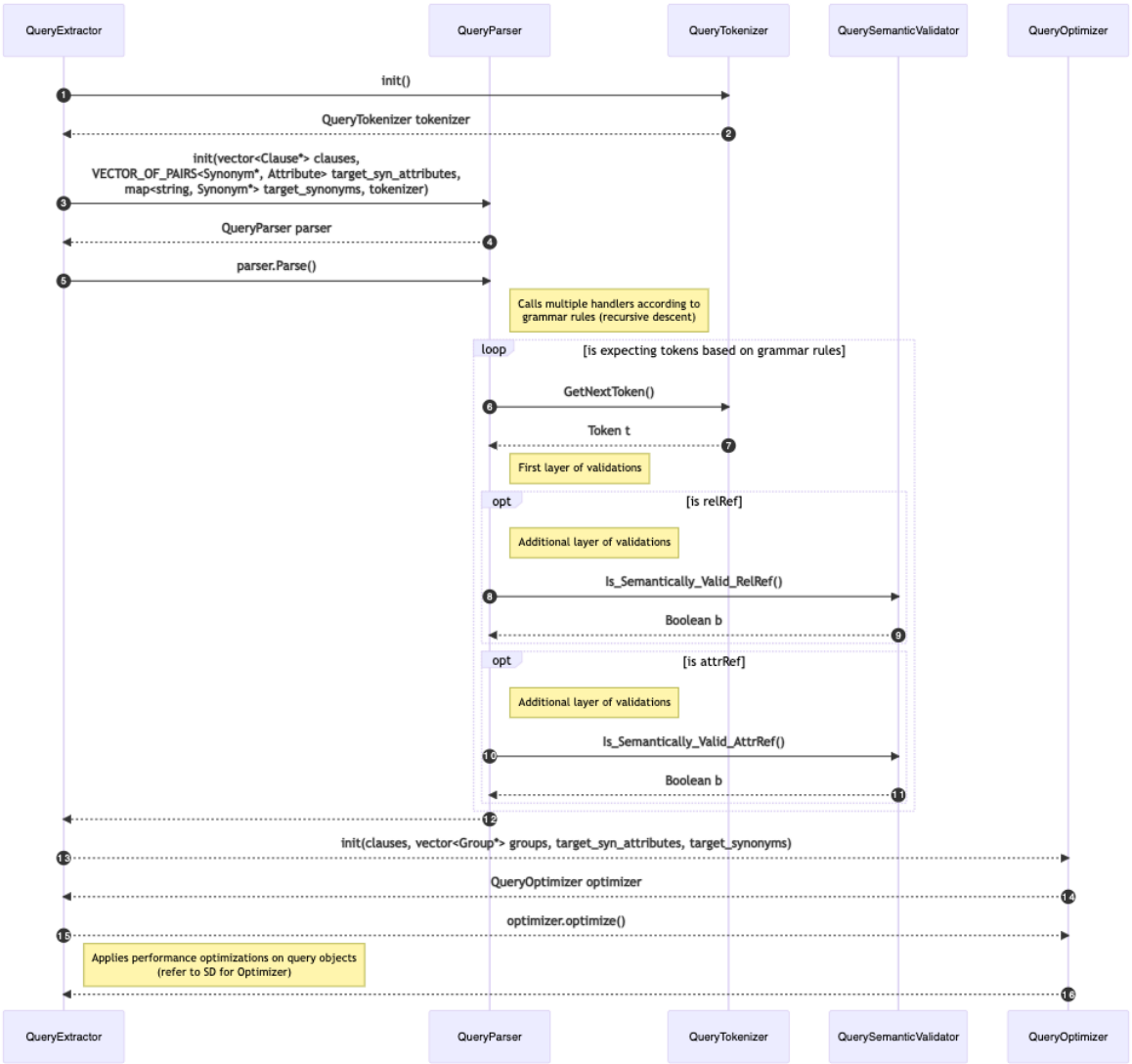


Figure 28 Query Extractor Sequence diagram

At a high-level, the QueryExtractor (ie query preprocessor) calls the QueryParser that parses the input query and populates the query objects with relevant

information. Subsequently, the QueryExtractor calls the QueryOptimizer to apply techniques (like grouping clauses) to increase efficiency of query evaluation. The subsequent portion is chronicled in the optimizations section 1.6.5.

Query Objects

The table below shows how we represent the extracted data. Note that the 'groups' data structure is the only one that is passed on from the query extractor to the query evaluator; other data structures below are only listed to enhance the contextual understanding of the reader. Auxiliary data structures are omitted.

Data structure type	Data structure name	Description
VECTOR<Clause*>	clauses	All the clauses in the query (which could be of type such that, while or pattern).
LIST<Synonym>	synonyms	All the valid synonyms in the declaration section of the query. Used internally within the query extractor.
VECTOR_OF_PAIRS <Synonym*, Attribute>	target_syn_attributes	The 'target' synonyms of the query, i.e. the synonyms that comes after the 'select' keyword, as well as its attribute if it was specified.
VECTOR<Group*>	groups	All the groups created from 'grouping' clauses together (for evaluation). The initial grouping may be improved by the optimizer.

Table 9 Description of Query Objects

Tokenizing, Parsing, Validation

The backend parser works similarly to the frontend parser, in that it uses a top-down parsing approach leveraging recursive descent. Our parser makes use of a tokenizer (QueryTokenizer), which uses string and regex matching to tokenize the input stream from start to end. The parser then syntactically validates the input stream of tokens against the expected grammar lexicons during recursive descent, populating the query objects in the process.

For semantic validation, another sub-component called QuerySemanticValidator is consulted. The following section describes the changes to designs and list them chronologically to discuss our decisions.

For iteration 1, the QuerySemanticValidator only contained logic for semantic validation for the arguments to the relationship references (relRef) for the “such that” clauses, as demonstrated in the figure below.

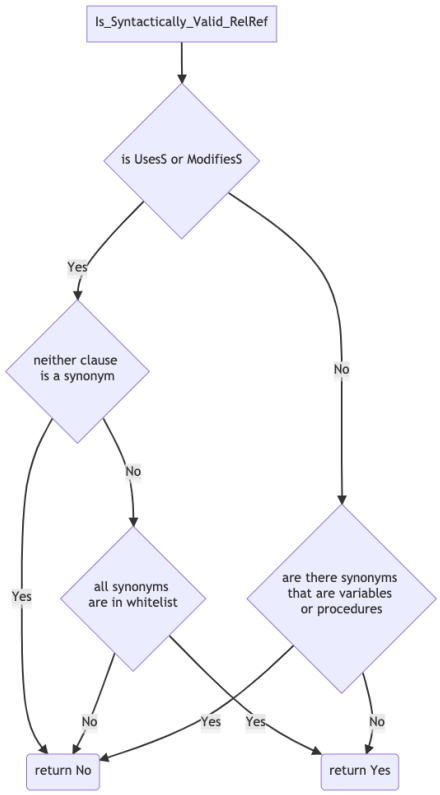


Figure 29 Flowchart for validating a relationship reference (RelRef)

As seen in the figure above, if the RelRef for the ‘such that‘ clause was either UsesS or ModifiesS, the key idea in iteration 1 was to ensure that any synonyms in the arguments must be in a whitelist. The whitelist comes from the semantic grammar of BasicSpa. For example, it is known that if a synonym appears in the righthand side, it must be a variable. Taking UsesS as an example, if a synonym appears in the lefthand side, it must either be assign, if, procedure, print, while, stmt or call. Similarly, there is a whitelist for ModifiesS. For the basic RelRefs (concerning relationships between statements), the arguments are semantically valid if synonyms (if any) are not variables or procedures.

However, we realised that this approach involved a lot of if-else logic, which was flexible but not as extendable. Therefore, for iteration 2 & 3, we used the idea of whitelisting by creating a mapping table (table-based validation), reducing the need for having if-else dispatching logic. We then utilised the similar design pattern leveraging table-based validation for validating the semantic validity of synonym-attributes. The general formats of the maps for relref and attrribute validation are as follows:

Table Usage	Table Prototype	Table Contents
Semantic validation of relref + combination of left and right arguments	Set<RelRef rf, DesignEntity lhs, DesignEntity rhs>	Combinations involving relrefs with 1 and 2 synonym arguments. This table does not need to consider the case where the relref has both non-synonym arguments (ie 0 synonyms).
Semantic validation of attribute based on synonym type	Set<Synonym syn, Attribute attr>	Combinations of design entities and acceptable attribute types. E.g. design entity of type ‘constant’ is mapped to attribute of type ‘value’. The table does not need to consider prog_line design entity due to restriction of pql grammar.

Table 10 Validation Strategy

1.7.2 Query Evaluation

After the Query Extractor is done with parsing and verification of the query, the relevant information regarding the query (as specified in Query Objects) will be passed to the QueryEvaluator via the QueryController. The QueryEvaluator makes use of a table data structure (QueryEvaluatorTable) to store the intermediate query results within each group. This shall be referred to as a group table.

The following architecture diagram provides an overview of the subcomponents within the QueryEvaluator component.

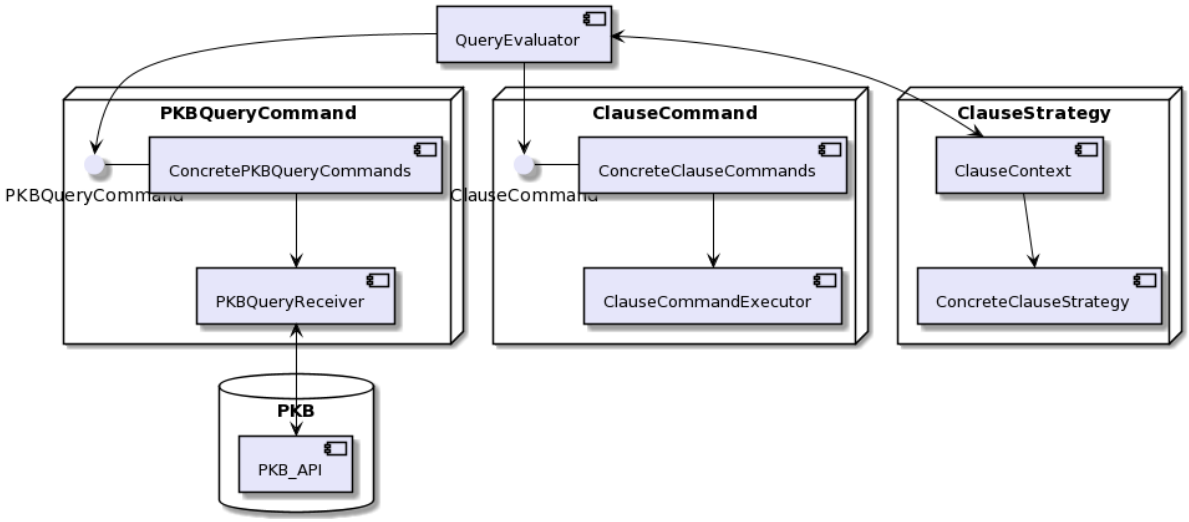


Figure 30 Activity Diagram for QueryEvaluator

From the above diagram, the QueryEvaluator follows the façade class design pattern by abstracting complex interactions between the various other subcomponents. The classes in the ClauseStrategy implements the strategy design pattern. Lastly, the PKBQueryCommand and ClauseCommand subcomponents implement the command design pattern.

Let us now analyze the API discovery process of interactions between the subcomponents and the PKB based on the type of clause being evaluated. This is done with the help of the sequence diagram below. Note that the following sequence diagram omits some information not relevant to how different clauses are evaluated, for brevity.

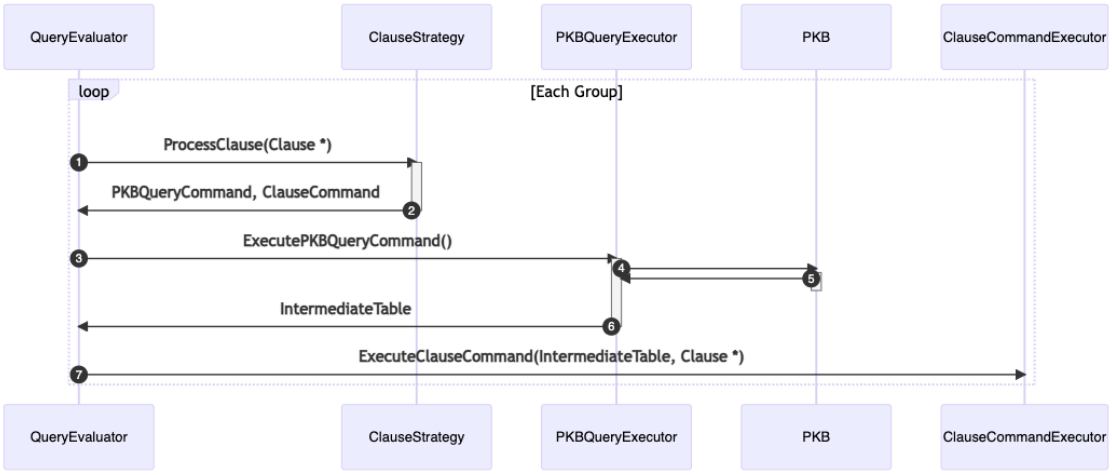


Figure 31 Sequence Diagram for QueryEvaluator

Each query, *q*, will consist of *t* target synonym (e.g. Select BOOLEAN or <a1, p1>) and *n* number of groups of *x* clauses, where $\forall n, \mathrm{t},\ \mathrm{x} \in Z_{\geq 0}$. Assuming there are b number of non-boolean group, and each will correspond to exactly 1 group table, then $b \leq t$ since a non-boolean group may contain more than 1 target synonym.

In all cases, the evaluation of the various clauses will follow the sequence diagram above. The ClauseContext will first determine if the type of the clause before calling the relevant ClauseStrategy, to process and determine the appropriate PKBQueryCommand and ClauseCommand to return. At this stage, the following class diagram helps illustrate the interaction between the various classes.

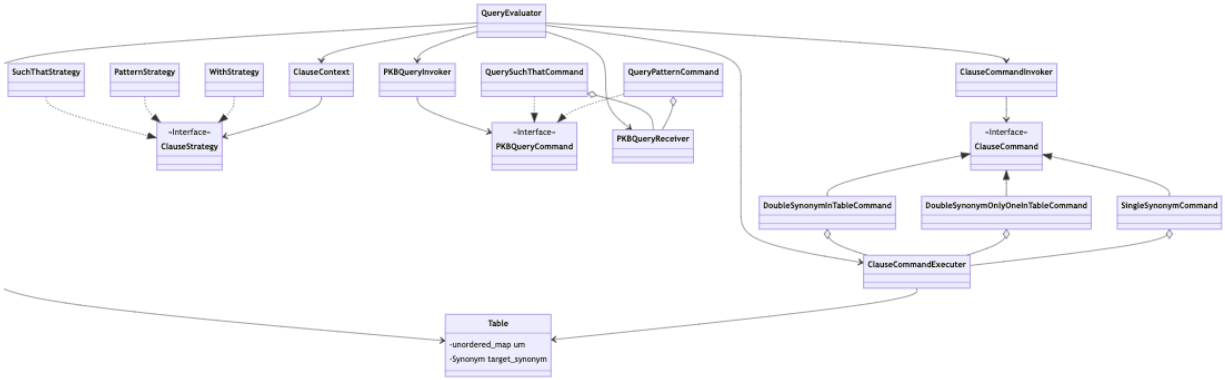


Figure 32 Class Diagram for QueryEvaluator

As seen from the class diagram and sequence diagram above, each component has been designed to help reduce coupling while maintaining a good level of cohesion. Each subcomponent's responsibility can thus be summarized in the following table.

Component	**Responsibility**
QueryEvaluator	Responsible for receiving the query information and coordinating all the subcomponents together. (Façade design pattern)

ClauseStrategy	Responsible for creating the appropriate PKBQueryCommand and ClauseCommands based on the clause being processed. (Strategy design pattern)
PKBQueryCommand	Responsible for the business logic of calling the PKB API based on the concrete command. (Command design pattern)
ClauseCommand	Responsible for the business logic of perfoming set logic on the IntermmediateTable and the group table based on the concrete command. (Command design pattern)

Table 11 Responsibilities of different components

Once all the groups of clauses have been evaluated, the QueryEvaluator will the pass the UnformattedQueryResult back to the QuerySystemController as shown in Figure 24. The activity diagram below summarizes how the QuerySystemController is populated with data and passed to the QuerySystemController (which will eventually pass the object to QueryProjector).

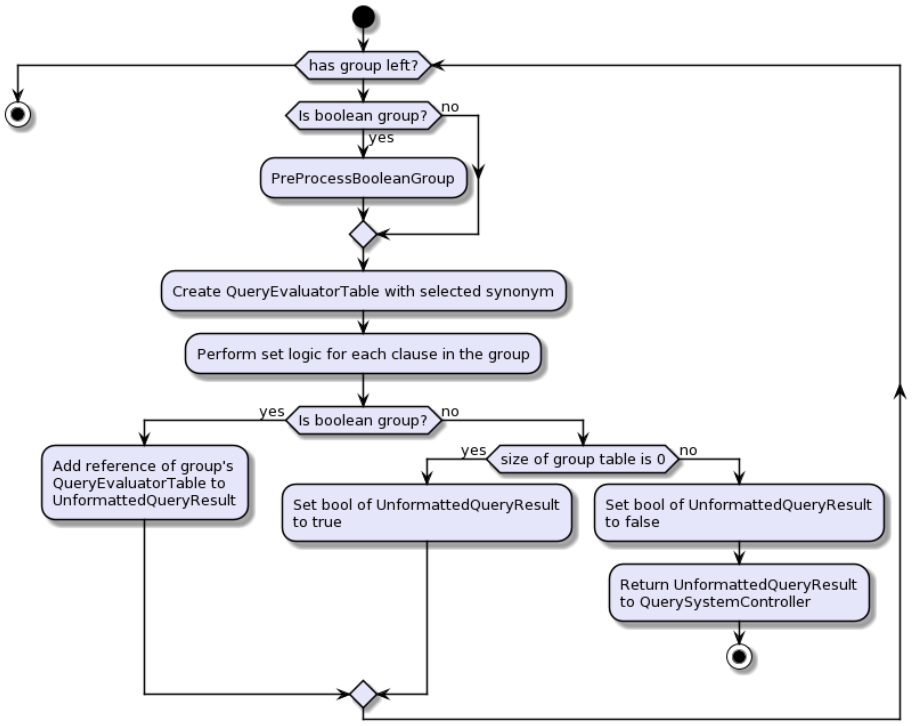


Figure 33 Activity Diagram for populating UnformattedQueryResult

1.7.3 Query Evaluator – PKB Interaction

Based on the table above, we can see that the only point of interaction between the QueryEvaluator and the PKB, would be the PKBQueryCommand.

Since the ClauseStrategy's only responsibility is to determine the type of the clause, the PKBQueryCommand will then be responsible for determining the correct PKBQueryReceiver method to be called, where each method would call at most 1 API belonging to the PKB. The following class diagram details the interaction

between QueryEvaluator and PKB; note that ‘pattern’ and ‘with’ clause works similarly and has been omitted for brevity.

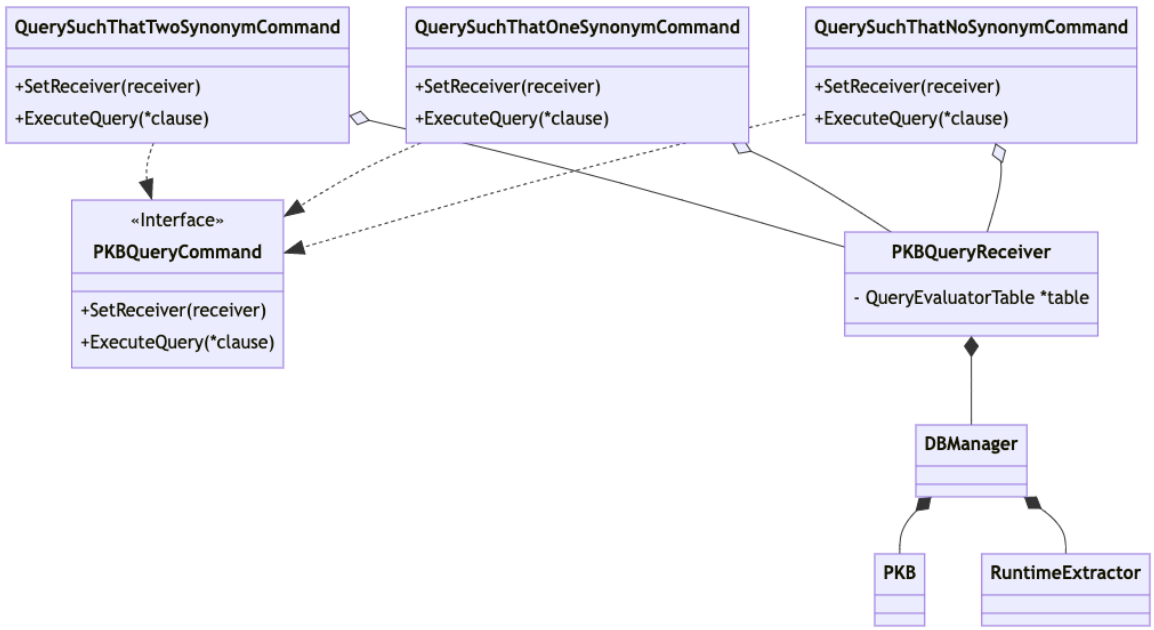


Figure 34 Class Diagram for QueryEvaluator - PKB interaction

The DBManager implements the Proxy design pattern to isolate the responsibility of calling the RuntimeExtractor, if required, due to the requirements of implementing runtime evaluation of information for certain clauses such as ‘affects’ as an example.

The following activity diagram illustrates the details of how PKBQueryCommand retrieves the required information from the PKB with reference the example source program (in 1.2) with the following query:

```
variable v; read r; print p; Select v such that Follows(r, p)
```

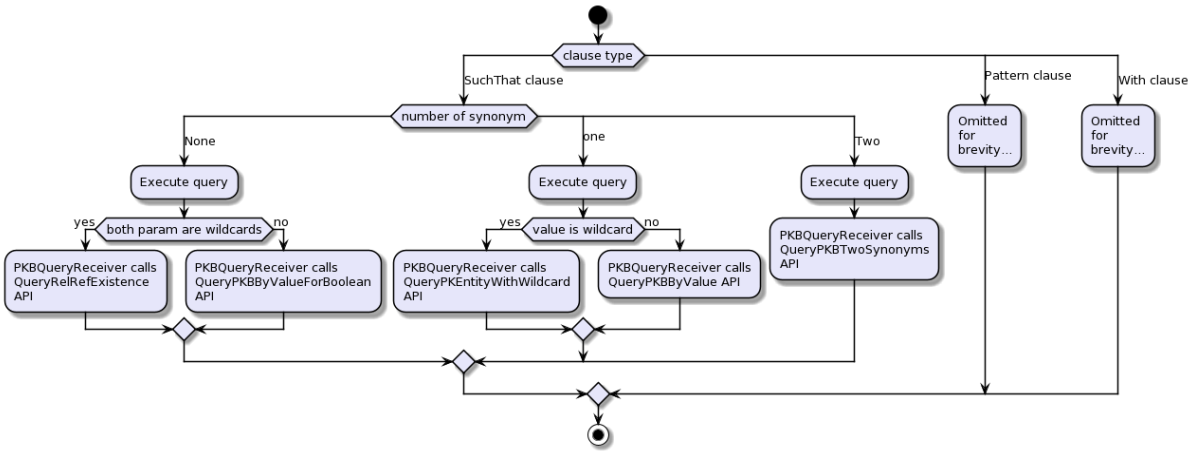


Figure 35 Activity Diagram to Determine PKBQueryCommand for SuchThat clause

1.7.4 Query Projection

The QueryProjector takes in an UnformattedQueryResult which contains the QueryEvaluatorTables to be formatted and projected. The QueryProjector does this by retrieving from the Tables the appropriate columns specified by the target synonyms. For results that require a tuple, the QueryProjector is responsible for performing cross product and formatting of the results in the Tables (including based on the desired return attribute type). Finally, it will send a list of strings to the TestWrapper to answer the query.

1.7.5 Query Optimization

Optimizations for the Query Object

As introduced in the sequence diagram at the beginning of the section 1.7.1, after successful parsing and creation of the query objects, QueryOptimizer performs optimizations, before they are sent for evaluation. The optimizations include division of clauses into multiple groups, sorting of groups for evaluation, and sorting of clauses with the group, as per the recommendations in the lecture notes. As a bonus, we also drop duplicate clauses to optimize certain edge case scenarios. These optimizations and their benefits are listed in the table below:

Optimization	Benefits	Comments
Dropping duplicate clauses in query	Reduces overhead of creating and processing duplicate clauses.	e.g. *stmt s1,s2;* *Select s1 such that Follows(s1, s2) and Follows(s1, s2)* In such a query, after we have seen the first clause, the duplicate second clause can be ignored immediately after the parsing phase and we do not need to insert it into the query object.
Grouping clauses with common synonyms together	Keeps intermediate table sizes small by reducing the need for cross products.	Our team implemented a custom grouping algorithm that might be of notable interest to readers. It is explained in further details below this table.
Reordering of Groups	Allows for early query termination if a ‘boolean’ group evaluates to false	‘boolean' groups (don’t contain a clause with target synonym) will be reordered to the front, and in order of size (number of clauses)

Reordering of Clauses within each Group	Clauses that are slower to evaluate, or have larger results should logically be evaluated last to improve worst-case performance.	Slower queries especially those with runtime evaluation (e.g. Affects, Affects*, Next*) are evaluated last. With clauses are evaluated before such that and pattern clauses. Clauses with lesser synonyms are evaluated first. Sort clauses such that at least 1 synonyms has been seen before (refer to explanation on grouping algorithm below)

Grouping logic

As mentioned in the table above, grouping of queries in an optimized way is important for performance reasons. Initially, for iteration 1, our QueryGrouper had a rudimentary implementation that could only group a maximum of one such that and one pattern clause together based on common target synonym, as shown in the following activity diagram.

For iteration 2 and 3, we reimplemented the algorithm to support groupings of any number of clauses, and any valid combinations of such that, with and pattern clauses. We provide readers with the following activity diagrams to summarize the key logic of the algorithm, followed by an explanation.

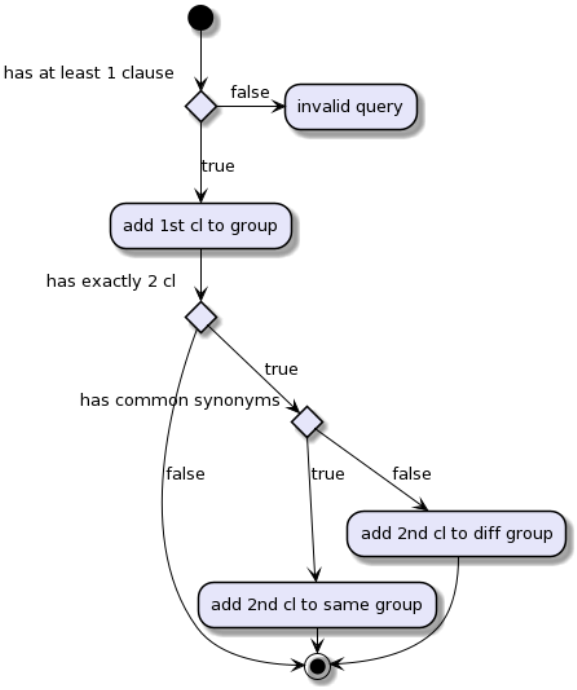


Figure 36 Activity Diagram for Grouping Logic (iteration 1)

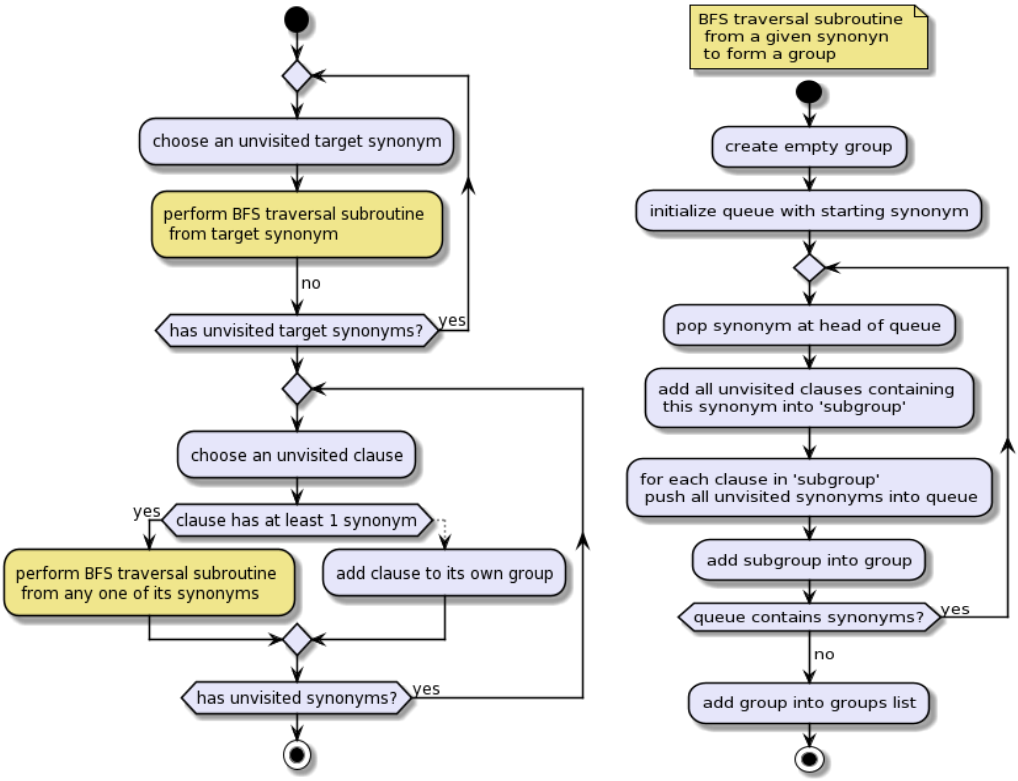


Figure 37 Activity Diagrams for Grouping Logic (Iteration 2)

As seen in the activity diagrams, we first perform a breadth-first search subroutine (BFS) starting from each unvisited target synonym, using a queue. For each synonym that we deque, we will create a subgroup that contains clauses that contains this synonym. If any of these clauses contain other unvisited synonyms, they are enqueued, and this process repeats till the queue is empty. Each subgroup will be appended to the Group object, which is returned to the main routine of the grouping algorithm. Finally, the main routine groups any unvisited clauses using the same approach, ensuring that all clauses have been assigned to a group.

The **design choice** to first group clauses into ‘subgroups’ is deliberate and is a **performance optimization**. Our algorithm guarantees that clauses in a subgroup share a common synonym. As mentioned in the table above, further optimization of clause order within a group is done for iteration 3. However, in the general case clauses will not be reordered across subgroups (unless they are runtime queries like affects/affects*/next*). This helps to avoid the scenario where evaluation of clauses (in the list-order they are given) results in an expensive encounter with two unseen synonyms, which will blow up the state space during table merging.

Example: Reordering of groups & clauses in query

```
assign a1, a2; while w1, w2; Select a2 pattern s2(“x”,_”x”_) such that
Modifies(s2, “y”) pattern a1(“x”,_) such that Affects(a1, a2) and Parent*(w2, a2)
and Parent*(w2, a2)
```

Final list of groups and clauses within them:

[[`pattern s2(“x”,_”x”_), Modifies(s2, “y”)`], [`pattern a1(“x”,_), Parent*(w2, a2), Affects(a1, a2)`]]

We note that a duplicate query was dropped after parsing. Also, the Boolean group containing clauses with non-target synonym s2 are ordered first. Within the second group with 3 clauses, we notice that pattern clause comes first, and the Affects clause is ordered last (expensive runtime query).

Clause Scoping

We know that most queries would probably include many clauses. As seen from the example above, this means that some synonyms would already be present in the table during evaluation. This provides us with an opportunity to optimize query evaluation during runtime by limiting the scope of values for the synonym(s) in a later clause. For clarity, let us compare how scoping improves PKB and QueryEvaluator’s runtime

Component	Without scoping	With scoping
PKB	Retrieve all possible entities given the DesignEntity, then retrieves the relevant information for the entities to be passed to the QueryEvaluator	Retrieves information only for the entities provided by the QueryEvaluator.
QueryEvaluator	The evaluator runs through the possible values given by the PKB to find a match, with some of the values being completely redundant.	The list of possible values will be shorter compared to no scoping, resulting in a faster evaluation.

Table 12 Comparison of Scoping and non-scoping for PKB and QueryEvaluator

Early Termination

As mentioned in the earlier section, Boolean groups are ordered first, this allows the QueryEvaluator to terminate early if any of the Boolean groups returns false and returns the UnformattedQueryResult to the QuerySystemController.

Another part where we can utilize early termination is if the QueryEvaluatorTable for any of the non-boolean groups has 0 rows. Since the order of the runtime clauses are prioritized last, this provides an opportunity to avoid runtime evaluation of costly clauses.

1.7.6 Design Decisions

Data structure for Queries

For the data structure for Queries, we were deciding between a AST of clauses and a vector of clauses (within groups). While both are equally viable ways of solving the same problem, we preferred to use the latter. This can be seen from the table below followed by a comparison of the weights of the pros and cons for each criterion.

Criterion	AST of clauses	Vector of Clauses (per Group)
Ability to distinguish Boolean groups from non-boolean groups	+ve: Easy to implement an additional variable to store the root node for Boolean clauses and distinguish Boolean and non-boolean ASTs. -ve: Might be difficult to implement the joining of nodes with common synonyms since we need to either traverse the AST or store the list of synonyms separately.	+ve: Easy to identity boolean groups and extensible when requirements change. -ve: Still requires effort on the part of the programmer as we require the group objects or clause objects to have a comparator, for reordering during optimization.
Retrieval of information should be efficient	-ve: Accessing data in an AST is less efficient compared to a vector	+ve: Accessing data in a vector is efficient with O(1) if the vector is already in the desired sorted order.
Ease of reordering of clauses	-ve: Difficult and costly to implement an algo for the reordering of the clauses	+ve: Easy to change the comparator for the group or clause object. However, if sorting is applied multiple times on a vector, it will incur overhead. As an extension, to support dynamic reordering efficiently, the same comparators can be easily input to a priority queue to update the order of the clauses.
Efficiency of creation of data structure	+ve: Creation of the AST not too expensive	-ve: The creation of a vector is rather fast. If the implementaion is extended to use a priority queue, the creation of the priority queue might be rather slow considering that the insertion of a new object will take O(lg N) time for each insertion (trade off with efficiency of creation/balancing vs efficiency of sorting)

Table 13 Comparison of data structure for query storage

Based on the evaluation of the benefits of each, we made use of the storing the clauses as a vector (in groups). The key benefit of this approach would be that it would avoid unnecessary complexity in dealing with the AST data structure, which would consume valuable development time.

It must be noted that in our current implementation as of iteration 3, due to time constraints, the reordering of the clauses is not repeatedly done dynamically, but only done initially before the query objects are passed to the evaluator. As an extension, our implementation can be easily modified to allow for multiple rounds of reordering. As mentioned in the table, the vector can be easily changed to a priority queue, and since we already implement a comparator for sorting, it can be fed to the priority queue. For more advanced optimizations, the comparator can be updated.

Query Validation

For the QuerySemanticValidator introduced in section 1.7.1, to summarize, even though table-based validation requires higher upfront time investment to construct on the part of the programmers, making use of table-based validation helps us to reduce having complicated if-else dispatch, improving extensibility. Furthermore, reducing if-else branching makes the code faster; mapping tables are also suitable as our project does not need to optimize for memory footprint. Our current implementation strikes a reasonable balance of making use of validation tables and some form of If-else logic to achieve our deliverables within the time frame of the project. to achieve our deliverables within the time frame of the project.

Optimization of Groupings & reordering of clauses

For the grouping algorithm introduced in section 1.7.5, to summarize, the tradeoff for having a slightly more complex grouping algorithm (based on the idea of subgroups) is the benefit that we avoid huge performance losses incurred due to merging tables involving 2 unseen synonyms. As mentioned, we supplemented this subgroup ordering with other heuristics for finer grained ordering of clauses and thought of handling edge cases such as repeated duplicate queries.

Query evaluation

For the QueryEvaluator, the process for the choice of design decisions were made based on the following problems and potential solutions

Problem	Solution
The number of target synonyms can be 0 or more than 1.	Each group table should be responsible for their own target synonyms.
Table manipulation should be abide by SRP but each clause has complex operations which each has a different way of calling the table manipulation.	The clause evaluator should be an invoker and the command pattern should be used with a table manipulator
The query to the PKB will be different for different types of clause.	Create a command design for clauses to query the PKB. This helps with keeping the implementation open to extension.

There are now too many commands to consider with the new clauses and features to be supported.	Use a strategy design pattern since each clause has similar features but different strategic approaches to create a suitable design.

Table 14 Problem and design solutions for QueryEvaluator

1.7.7 Abstract API

API	**Description**
Query Extractor	
VOID ExtractQuery (QUERY query)	Receives a QUERY object (likely string), which may be a valid or invalid query.
Query Evaluator	
UNFORMATTED_QUERY_RESULT EvaluateQuery(GROUP_LIST groups)	Evaluates the information passed to the QueryEvaluator upon the object's creation. It will then return the list of possible result values.
QueryProjector	
FINAL_FORMATTED_RESULT FormatQuery(UnformattedQueryResult unformatted_result)	Receives raw results of query evaluation as a custom wrapper object of type UnformattedQueryResult, and formats them before passing the list results to the caller (TestWrapper).

Table 15 Important abstract APIs for Query Processor

2. Testing

Systematic testing proved to be significantly helpful in managing our development workload and speed.

We implemented unit tests along with feature development and that helped us guarantee the meeting of abstract API requirements. Unit testing also ended up being a starting point for Code review sessions, where we simply had to see what unit tests the implementor had written to get a progress update for that implementation. Regression testing was automated thanks to good code coverage of the unit tests. Lastly, we found unit testing to serve as a basic building block on which integration test cases could be based on.

Integration testing was done for two main purposes: to integrate the subcomponents (e.g., within the SourceProcessor) as well as to integrate the various larger components (e.g., SourceProcessor to PKB, PKB to Query System). Integration testing across components gave us the opportunity to ascertain the progress of subgroups and understand the unsupported features and backlogs as well. Additionally, Integration testing provided also gave us the opportunity to use different testing environments (compiler toolchains) which highlighted areas of improvement to make our code truly cross-platform. For example, a testcase may pass locally (on Windows10, MSVC toolchain) but fail on Fedora CI environment or on MacOS because of the different default optimization settings of our differing compilers.

2.1 Approach to Test case design

In general, for all types of testing, the main goal was to come up with quality test cases because excessive quantity of testcases can mean regression in test cases themselves should there be major changes to our codebase. Here are some steps we followed:

1. Identify the Design Abstractions and relationships to be tested.
2. Identify the equivalence partitions (EP) for these to avoid unnecessary testing within a partition. This gives good coverage without redundant test cases.
3. Create positive test cases for the intended functionality to test if these basic cases are supported.
4. Create the negative cases based on the EPs as well.
5. Create edge cases afterwards. To come up with these edge cases, an adversarial approach works best since the goal would be to break the intended functionality of the feature.
6. Experience based testing: Create test cases based on previous experience or knowledge of the behaviour of the program.
7. Have someone not part of the implementing team to create edge cases (black box testing)

2.2 Unit testing

In this section, we outline some examples of test cases created for different parts of our program.

2.2.1 Unit tests for PKB

The input to the PKB is a Deliverable object (containing hash maps of all entities and relationships). The PKB has numerous outputs, ranging from Boolean to vectors of entities to vectors of entity pairs. We decided to use the actual Deliverable class in our unit tests instead of creating a Deliverable stub due to how coupled the PKB is with the Deliverable class as well as how simple the Deliverable class is (mostly consisting of various getters and setters).

Our unit tests therefore test that the PKB is functioning correctly by first populating it with a Deliverable object and then retrieving the various entities/relationships from it. The information retrieved from the PKB must match exactly the information passed to it through the Deliverable object.

1. Retrieval of entities from PKB

Test purpose	Verify that entities can be retrieved from the PKB after correct population

Required test input	Deliverable object representing the following SIMPLE code:
	`// SIMPLE code`
	`procedure p1 {`
	`read v1;`
	`v1 = 1 + 2 + v2;`
	`if (x < 3) then {`
	`while (x > 4) {`
	`print v1;`
	`v1 = v1 - 1;`
	`}`
	`}`
	`}`

Expected results	Complete vectors of each different kind of entity: Procedure, Read, Assign, Variable, Constant, If, While and Print.
Requirements	Deliverable object

2. Retrieval of relationships from PKB

Test purpose	Verify that relationships can be retrieved from the PKB after correct population
Required test input	Same as above

Expected results	Querying the different relationships returns all relationships in their expected formats. For example, querying PKB::GetFollows(“1”) should return a vector consisting of a single AssignEntity (representing statement 2). All relationships and methods are tested in a similar manner.
Requirements	Deliverable object

2.2.2 Unit tests for Query Processor

Due to brevity, we will illustrate 1 unit test from the Query Extractor and Query Evaluator respectively.For the Query Extractor, the unit test highlighted in the table below tests that input query is correctly parsed and correct clause objects are created (QueryParser).

Test purpose	Verify Clause objects created for valid query with 3 clauses of type such that, pattern, with
Required test input	`assign a; stmt s1; prog_line n; while w;` `Select p such that Parent*(w,a) with n = s1.stmt# pattern a(“y”, n)`
Expected results	The vector of Clauses after parsing should contain 3 clause objects, of type such that, with and pattern respectively with the correct fields populated. (order of clauses is not important for this test case, but we can expect them to be in the order they appeared in before any reorderings).

For the Query Evaluator, the unit test we chose to highlight is related to the ClauseStrategy subcomponent. In general, the unit tests helped ascertain the correct Command objects were being created for each type of clause. The table below summarizes 1 such unit test.

Test purpose	Verify Command objects created for a such that clause with 2 synonyms
Required test input	(Mock)QueryEvaluatorTable, SuchThat - clause
Expected results	PKBQueryCommand = QuerySuchThatTwoSynonymCommand, ClauseCommand = DoubleSynonymBothPresentCommand
Requirements	The MockQueryEvaluatorTable class was created so that the ClauseStrategy subcomponent could be tested in isolation.

2.3 Integration testing

Integration testing is a necessary and important part of our project as it provides the guarantee that our individual components can be linked with one another. In our team, the delegation of tasks was such that every person worked on separate components, with little to no overlap between sections directly coded. While unit testing ensures that individual components can work in isolation from one another, it is still necessary to ensure that the inputs, outputs, or side effects of one component match up with those of another. At the same time, integration testing serves as a ‘living’ documentation for how components are supposed to interface with one another.

We broadly separated our integration testing into two main parts: testing the Source Processor to PKB link as well as the PKB to QueryProcessor link. The logic behind this decision is to adopt a stepwise approach such that bugs that arise are easier to pinpoint and address.

2.3.1 Source Processor to PKB

The integration between the Source Processor and the PKB can be broadly classified as the parsing and database population segments of our static program analyzer. In the context of this integration test, this would refer to input SIMPLE files to the Source Processor being completely and adequately represented by information retrieved from the PKB. Our approach was therefore to create a representative SIMPLE file to be parsed and thereafter ensure that information retrieved from the PKB not only matched but fully represented all relevant features of the input file. To this end, we used the same SIMPLE code as in 2.2.1 Unit tests for PKB, as it is a completely representative SIMPLE program for Iteration 1. We then verified that each individual entity and relationship was stored correctly within the PKB.

2.3.2 PKB to Query Processor

The integration between the PKB and the Query Processor components can be broadly classified as the evaluation segment of our static program analyser. Since this part only relates to the pipeline including and after the PKB, we can assume that information passed to the PKB is accurate. We therefore must determine only whether the PKB- Query Processor components are able to evaluate any given query, given the query as well as the data structure that the PKB takes in to be populated (Deliverable).

As SIMPLE source code parsing is not in the purview of this integration test, we created only a singular Deliverable object representative of the entire suite of Iteration 1 requirements. We then created a list of queries that could adequately test the query evaluating functionality. The Deliverable object was based on the same SIMPLE code as in 2.2.1 Unit tests for PKB, and in fact is the same object. This is because the object used in 2.2.1 is already completely representative of the different features within the scope of Iteration 1. The queries used are as follows:

```
// Query 1:
variable v;
Select v
// Query 2:
assign a;
Select a
// Query 3:
read r;
Select r
// Query 4:
procedure p;
Select p
```

```
// Query 5:
assign a, b;
Select a
// Query 6:
read r; assign a;
Select r such that Follows(r, a)
// Query 7:
assign procedure;
Select procedure
// Query 8:
read r; assign a;
Select p such that Follows(r, a)
// Query 9:
assign a; assign b;
Select s such that Follows(a, b)
// Query 10:
variable v; read r; assign a;
Select v such that Follows(a, r)
// Query 11:
variable v; read r; if ifs;
Select v such that Follows*(r, ifs)
```

We believe that this set of queries is it encompasses all the different types of evaluations. Namely, these include basic lookup of each unique type of object in the PKB, basic relationship lookup, reverse relationship lookup, evaluation in which the clause resolves to a Boolean (True and False) and transitive relationship lookup. As different relationships are stored and retrieved in the PKB using the same data structures (e.g. all transitive relationships stored the same way), there is no need to test them individually since they belong to the same equivalence partition. Different edge cases also exist within the test set, for example keywords being used as names (procedure) and multiple declarations (assign a, b).

2.4 System testing

In system testing, our goal is to ensure that the query output will correspond to a provided SIMPLE program input and fulfil SPA requirements. It is an extension of integration tests and should efficiently and completely cover the requirements.

In designing test cases, we identify the equivalence partitions for both input SIMPLE program and queries separately. This is to ensure that the tests give good coverage without missing out any. The types of queries created will be used across multiple test cases with differing input SIMPLE program to guarantee the accuracy of the parsing. Thereafter, some time is spent on negative and edge cases that can occur when the input SIMPLE program and the queries are both present in testing.

2.4.1 Choice of SIMPLE program

Our team came up with 3 main SIMPLE source programs to test our test suite of queries. The first is a relatively simple source program generated during the basic SPA implementation; the second is a more complex source program designed to

test the introduction of multiple levels of nesting. The last source program is the most complex program with multiple levels of nesting and utilizes EPs to ensure that every grammatical aspect of SIMPLE is covered. This incremental difficulty level ensures that debugging is manageable and incremental.

```
procedure p {
    x = z * 3 + 2 * y;
    print
    me;
    i = i - 1;
    x = x + 1;
    z = x + z;
    z = 1;
    read
    me;
    z = z + x + i;
    z = z - z + 2 * z % z;
}
```

Procedure p was created to test very basic relationships to ensure that our program worked for most of the basic SPA requirements.

```
procedure Bumblebee3 {
    read
    x;
    y = 2 * z;
    z = z - 1;

    while (z > 0) {
        while ((nice + 1) > 12) {
            y = x * q - 5;
            while (x < 69) {
                z = z - 1;
                while (sleepTime > 0) {
                    while ((bananas * cherries) == 0) {

                        y = x + y * z + p * q;
                    }
                    if (q != z) then {
                        x = 1 ;
                    } else {
                        print x;
                        read x;
                    }
                    y = x + y * z + p * q;
                }
            }
            i = x + j + z;
        }
        if (x != z) then {
            read x;
        } else {
            read x;
            print x;
        }
    }
    print p;
    read p;
}
```

This test input tests for valid syntax validation as well as valid query processing. For example, “`print print; Select print`” can test for valid query processing of similar keywords and synonyms, while also testing for accurate relationships extracted.

```
procedure First {
    read x;
    read z;
    call Second;
}

procedure Second {
    x = 0;
    i = 5;
    while ((! (d<=j)) || ((6<d) && ((x>Third) && (2<=0)))) {
        x = x + 2 * y;
        call Third;
        while ((i!=i) || (! (2>d))) {
            if (x >= y) then {
                read x;
                v = 5;
            } else {
                y = x * z + 10 * x;
                print Third;
                read First;
            }
            print x;
            z = x % 1;
        }
        i = i - 1;
    }
    if (x == 1) then {
        while (x < 0) {
            print x;
        }
        if (!(z == 2)) then {
            print y;
        } else {
            i = (a / (26));
        }
        x = x+1;
    } else {
        z = 1;
        while (x < 0) {
            print x;
        }
        if (!(z == 2)) then {
            print y;
        } else {
            i = (a / (26));
        }
    }
    z = z + x + i;
    y = z + 2;
}
procedure Third {
    z = 5;
    v = z;
    print v;
}
```

This last test case is one of the most complex SIMPLE program used across our test suite. It was generated with EPs for nesting with complex expressions.

2.4.2 Design of Query test cases

For the choice of positive query test cases, we followed the following steps for the generation of test cases as outlined in section 2.1. However, we require some augmentation to those steps to achieve a more efficient and complete set of test cases.

Firstly, the heuristic we applied for the generation of test cases involved listing out the Equivalence Partition(EP) for each of the parameters in the clause. For e.g. 'such that' clauses has 2 parameters: `Calls(param1, param2)` whereas 'pattern' clauses has 3 parameters: `Pattern param1 (param2, param3)`. Below are the steps to generate the system test cases.

1. After generating the EP, we pick an actual value to be used within each EP.
2. Ensure that each value must appear at least once.
3. Ensure that each value must appear in at least one positive test case.
4. If there are more than one negative value in a negative test case, split them up such that there is at most 1 negative value in each negative test case.

Note that once the values were selected, it was relatively quick to generate these test cases with the help of a python script.

Secondly, while the above augmentation steps enforce a good coverage of positive system test cases, we require a second augmentation of steps for the generation of a comprehensive negative test case suite. Therefore, the generation of negative test cases were broken down into semantic and syntactic errors.

Semantic errors refer to queries which follows the grammatical rules, but does not make sense; For e.g., `print p; Select p Pattern p (_, _).` This does not work since only assignment, if, or while statements can follow Pattern.

Syntactic errors refer to queries which does not follow the grammatical rules; For e.g., `print p1; Select p1 Such That Follows(p)`. This does not work since RelRefs requires 2 parameters.

For the combination of clauses, we used a cross product to ensure that all cases are covered for queries with 2 clauses. Furthermore, we categorized queries with 1 clause and no clause to be of different behaviors and hence resulted in 3 over-arching test suites (not including edge/weird cases) for positive and negative system tests for queries.

To extend the testing for iteration 2 and 3, we continued to first test for single clause correctness of the new relationships as well as combinations of queries with 2 clauses. As an arbitrary number of clauses are expected in iteration 2 and 3, we then shifted our focus towards ensuring that our system can yield accurate results when merging multiple intermediate results during evaluation, and that it is able to project the correct results to the user. Conceptually, our general testing involved

crossing partitions where there are multiple clauses with different numbers of common synonyms (thereby affecting their grouping), with the partitions for the return types (e.g. BOOLEAN, 1 target synonym, multiple target synonyms with and without attribute). For return types, we also consider the different cases for synonym-attributes to ensure that the system projects the correct attributes to the user.

2.4.3 Additional SIMPLE source programs

Note that despite having 3 main SIMPLE programs tested against our query test suite, this omits test cases where we want to isolate queries in a program with only call, print, or read only statements, to ensure that all information is captured by the Source processor correctly. As such, additional programs included here illustrates provides a more complete system test.

```
procedure First
{
   print x;
   print y;
   calls Second;
}

procedure Second
{
   print z;
   print v;
}
```

```
procedure First
{
   calls Second;
}

procedure Second
{
   read z;
}
```

The main idea of these tests are to ensure that each DesignEntity will be accounted for in isolation.

2.4.4 Sample test cases

The sample query below can be applied to any of the SIMPLE source program detailed above in Section 2.4.1.

Test purpose	Test suite's purpose: Ensure the source processor extracts 'with' clause information correctly. Test query's purpose: Ensure that the processor is able to extract the attribute information for procedure and read correctly.

Test input And Expected test result	4 - synonym cross stmt s1, s2; assign a1, a2; call c1, c2; print p1, p2; read r1, r2; procedure pr1, pr2; while w1, w2; if if1, if2; prog_line pl1, pl2; variable v1, v2; constant const1, const2; Select pr1 with pr1.procName = r1.varName First 5000
Requirements	Requires Autotester

Test purpose	Test suite's purpose: Ensure the query evaluator performs the table manipulation correctly. Test query's purpose: Ensure that the query processor can handle queries with a 'circular' relation.
Test input And Expected test result	4 - Only such that clause, valid result, with circular relation stmt s1; assign a1, a2; variable v1; procedure pr1; prog_line pl; Select s1 such that Uses(s1, v1) and Parent*(s1, a2) and Modifies(a2, v1) 5, 7, 10 5000
Requirements	Requires Autotester

2.5 Stress testing

Our team utilized a python program to automatically generate a program with complex expression to stress test the query evaluator for queries which require more than 15000 rows of entries. This enabled us to ascertain if our data structures are efficient enough of the types of operations (insertion, deletion, etc).

In addition, this method of testing proved useful when our team utilized the clion profiler in conjunction with the stress tests. For instance, we managed to spot certain inefficiencies in data structure choices such as map over unordered_map. Even if the data structure is appropriate, there are some inefficient methods being used that can be spot by the profiler, such as vector.insert over vector.push_back, as an example.

3. Iteration 3 Extension

Implementation changes

The extension is completed up to NextBip and NextBip* for general CFGBip of extension b. With the creation of the RuntimeExtractor, it is easy to extend it by implementing the RuntimeColleague interface on a new extractor. Hence, no changes were made to any design entity already stored in the PKB, nor were any new components created except for the classes containing the extraction logic. These classes are namely NextBipExtractor and NextBipTExtractor. In a similar fashion, we have created AffectsBipExtractor and AffectsBipTExtractor, albeit it redirects to the non-bip version during execution.

Algorithm used

For NextBip, the Next relationship hash-map is first used to temporarily populate the NextBip hash-map. As with the other runtime extractors, there is a hash-map mapping an Entity to its related Entities as well as for the reverse relationship. For each Calls relationship, the Next relationship is removed and NextBip of the Call Statement and the called procedure's first statement is added, and the NextBip of the last statements and the statement following the Call Statement is added to the hashmap.

For NextBip*, DFS is done for every NextBip relationship to ensure that relationships in the cyclic graph will be correctly recorded. The ease of implementation was a key consideration for the algorithm chosen.

Since the relationship hash-maps are fully populated in one call, it can be populated in the PKB via `PopulateRelationships`. All subsequent queries will be directed to the PKB to use its data structures.

Abstract API

The abstract API is as per the RuntimeColleague interface written in the Runtime Extractor API section.

System testing

```
procedure n4 {
          if (reset == true) then {
                    alpha = 1;
                    call n5;
          } else {
                    call n5;
          }
}
procedure n5 {
    while (false == no) {
          print alpha;
          print beta;
          print charlie;
    }
}
procedure n6 {
          read beta;
```

```
}
procedure n1 {
          if (zebra > lion) then {
                    call n3;
                    while (lion > tiger) {
                              charlie = charlie % delta;
                              call n2;
                              delta = charlie % alpha;
                    }
          } else {
                    null = alpha;
                    call n5;
                    read delta;
          }
          charlie = delta;
     if (rit == esh) then {
          while (submodule == submodule) {
                    while (!(submodule == submodule)) {
                              null = null;
                              call n4;
                    }
          }
     } else {
         g = g;
     }
}
procedure n2 {
     call n5;
}
procedure n3 {
     call n4;
     call n5;
     call n6;
}
procedure no {
     read no;
}
```

```
stmt s1; assign a1;
Select s1 such that NextBip(s1, a1)
1, 12, 5, 10, 18, 22, 20
```

This test case tests for the proper linking of NextBip from the Call statement #15, which is Next to an Assign Statement #16, to procedure n2 to procedure n5. Hence, statement #5 should be in the answer.

```
stmt s1;
Select s1 such that NextBip*(9, s1)
1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29
```

This test case tests the linking of NextBip* of procedures being called multiple times and statement #9 will reach all other procedures but the last one.

4. Planning

Component	Team Member	Allocation
Source Processor	Wei Jie	- Reading of source to delimiter, and passing each statement to PSubsystem. - Creation of PSubsystem which adds trivial relationship and keep track of statement list. - Creation of Entity and Statement classes. - Creation of AssignmentExpression and ConditionalExpression. - Creation of Blocks and CFG - Creation of AffectsTExtraction - Linking of AffectsExtraction
	Ritesh	- Creation of Lexer (Tokenizer class) - Creation of RegexPatterns class - Creation of Logger - Creation of Token Datatypes and common utils functions - Syntax Validation - Creation of Clusters - HasAffects traversal
	Junxue	- Creation of EntityFactory - Creation of DesignExtractor and all Extractor subclasses (TransitiveExtractor/ VariableTExtractor/Next Extractors) - Creation of EntityUtil test class - Updating QueryProjector with set logic for tuples - Creation of DBManager, RuntimeExtractor - Implement NextT, NextBip and NextBipTExtractor
PKB	Oliver	- Creation of internal data structures to store various types of entities and relationships - Creation of populate methods to populate internal data structures with correct structures - Creation of ‘get’ and ‘has’ methods for all entities and relationships in the PKB - Write up for spa extension proposal.
Query Processor	Hemanshu	- Creation of controller for interaction with Query Processing system - Creation of backend Tokenizer - Creation of backend parser - Syntactic/semantic validation logic

		- Creation and population of query objects - Creation of grouping algorithm for related clauses, clause & reordering optimization.
	Max	- Creation of table data structure - Creation of query type data structure - Processing of various clause type - Processing of boolean and non-boolean groups - Overall design pattern planning and implementation for QueryEvaluator - Manipulation of the table data structure - Optimization planning and design - System test case design and heuristics

Table 16 Team Plan & Allocation

Task	Activity
Iteration 1	
Parsing source code	Parsing
	Tokenization
	Syntax Validation
	Entity Extraction
Relationship extraction	Non-transitive relationship extraction
	Transitive relationship extraction
Data population	Entities
	Non-transitive relationship
	Transitive relationship
Query Extraction	Tokenization
	Parsing
	Validation
Query Evaluation	Such that clauses
	Pattern clauses
Query Projection	API for formatting query answers
Testing	Unit, integration, and system testing
Debug	System testing
Iteration 2	
Supporting full spa syntax	Parse multiple procedures
	Support new lexicons and advanced relation types
	Support queries for Boolean and tuples
	Support multiple clauses
Refactorization	Sequence and API planning
	Refactorization of QueryEvaluator
	Creating new classes for design patterns
	Refactorization of QueryProcessor

	Refactorization of PKB API
Optimization	Optimize PKB data structures
Relationship extraction	Implementing Calls/Calls*
	Implementing Next/Next*
CFG creation	Implementing Blocks
	Implementing Clusters
Testing	Unit, integration, and system testing
Iteration 3	
Supporting full spa syntax (continued)	Support With clause & Attributes
Refactorization	Refactorize of PKB to use DBManager
	Refactorize RuntimeExtractor to use Mediator pattern
Relationship extraction	Implementing Affects/Affects*
Extension	Implementing NextBip/NextBip*
Optimization	Reordering of clauses & groups in Query Subsystem
	Scoped Querying
Testing	Unit, integration, Systems & Stress Testing

Table 17 Table of tasks

Our team made full use of GitHub's issue tracker for task creation and tracking. The table above contains a thorough list of our issues planned and created over the course of the iterations. For an even more comprehensive list, please refer here.

4.1 Tools setup and workflow

Communication

For our project, we decided to use the following communication channels: Zoom, Discord as well as Telegram. Zoom was used as a proxy face-to-face interaction in lieu of meeting in person. Telegram was used mostly for administrative purposes, such as setting meeting times. Our main discussion platform was Discord, chosen for the ability to create different channels for different agendas.

Project Management

GitHub Projects was our choice tool. We created specific workflows for our development process. Every issue was tracked, with issue templates to structure the conversation. We curated a list of labels (ranging from priority to relevant sub-team to discussion types) as well as milestones, each corresponding to a specific sprint. Each issue was assigned a number of these labels as well as which sprint it belonged to. We also made use of project boards, to easily gleam what issues were active at any one point in time. Finally, we relied on PRs for a best-effort code-review process in order to track and communicate our progress to each other asynchronously as well as learn.

Branch Organisation

We adopted a workflow in which we made separate branch folders for different categories. Namely, these categories included the subteam (‘pkb’, ‘SourceProc’, ‘PQL’), as well as a ‘docs’ folder and a ‘bugfix’ folder. All changes must be made on a separate branch from master, with a subsequent pull request to merge the changes. To enforce this, our master branch was protected, and all PRs required 3 reviews to be merged. Near the end, we also created a ‘staging’ branch, meant for debugging issues that arose during the testing phases.

Testing

For the creation of test cases (both SIMPLE source as well as queries), we made use of HackMD, for its collaborative features. The documents we created on HackMD were automatically synced to our GitHub repository. The use of python scripts for test case generation as well as convenience scripts written in bash for custom CI workflows helped us a lot as well.

Continuous Integration

CI was set up through Github Actions to automate testing for unit, integration, and system tests. This vastly improved the speed of regression testing at any stage of our project. It also sets up testing in our target environment. Refer to 5 Test Strategy for more information. Further usage of CI in the future can include cpplint to check on coding standards that we established (6 Coding Standards).

4.2 Gantt Chart

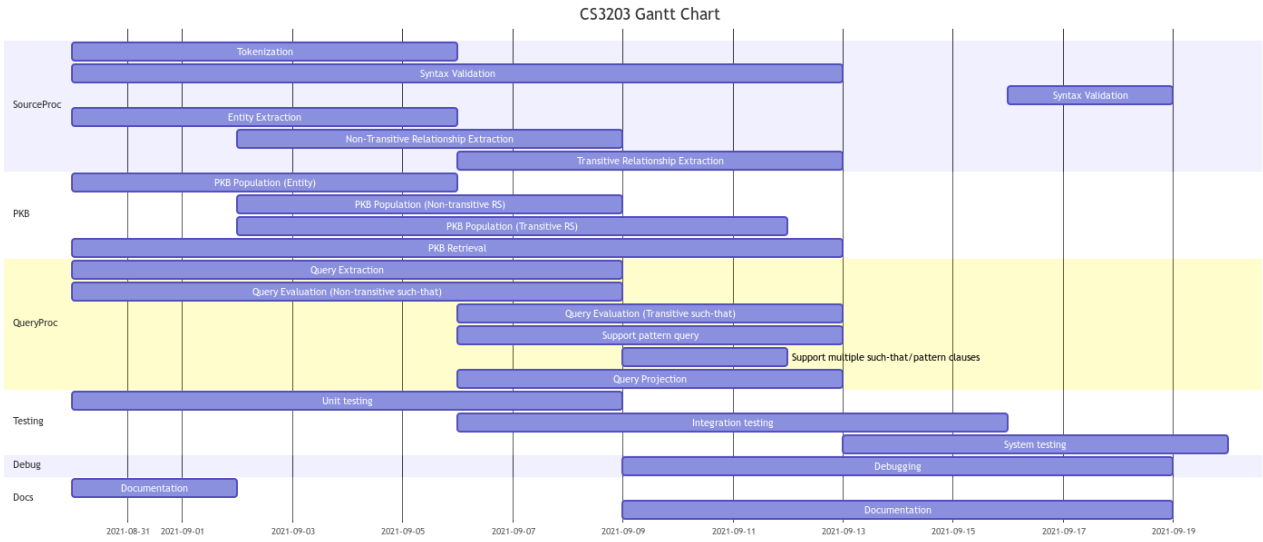


Figure 38 Iteration 1 Gantt Chart

We separated our Iteration 1 development into three distinct mini-iterations, or sprints. As can be seen from our Gantt chart, our strategy for developing the Static

Program Analyser involved first building the most basic components and subsequently creating more advanced features, all the while performing the necessary testing and documentation. We made a distinction between relationships that are non-transitive (e.g. Follows, Parent, etc.) with those that are transitive (e.g. Follows*, Uses, Modifies, etc.).

Sprint planning occurs every week on reserved Sunday and Monday evenings.

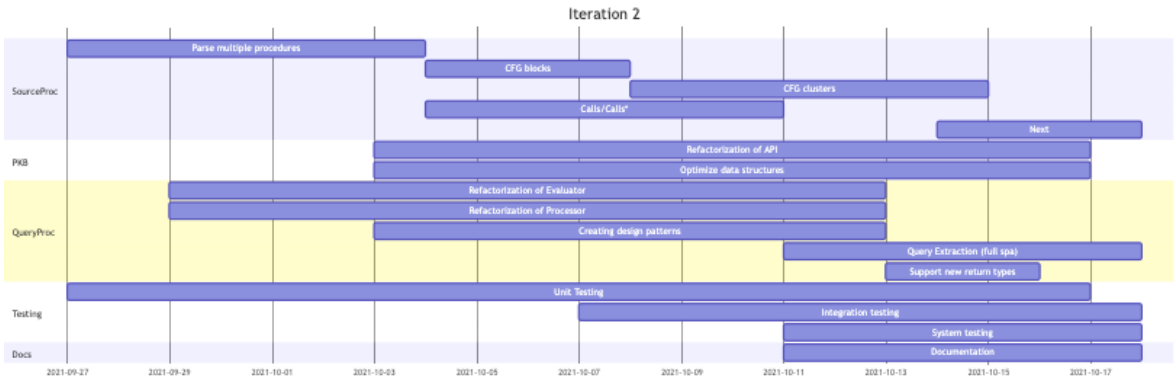


Figure 39 Iteration 2 Gantt Chart

For Iteration 2, the time is separated into 2 mini iterations only, iteration 2.1 from week 7 to 8 and iteration 2.2 on week 9. This is due to midterms occurring around week 7 and 8.

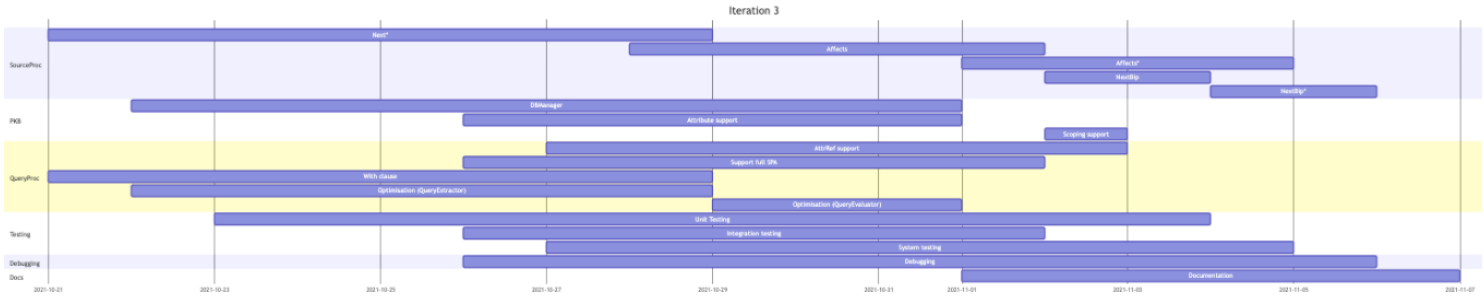


Figure 40 Iteration 3 Gantt chart

For iteration 3, our team focused mainly on completing the remaining features as well as system test planning and creation. This ensures that immediately after unit testing (or integration testing) is done, we can immediately run add the relevant system test into the CI. At this stage, the team began bug triage and assigning for

debugging. The last part of the iteration was then spent mainly on debugging and documentation.

5. Test Strategy & Bug Finding

5.1 Testing automation

To ensure that there is a guarantee of correctness within the repository, our team utilized continuous integration (CI) to automate testing whenever a commit is made into the master branch, or any pull request was created.

For the test suites, the SIMPLE source was automated using a python script for generation. As for the test suites, once the Equivalent partitions(EPs) are listed, a python script is also utilized to generate the queries.

The following sections provides more details for these 2 major sections

5.1.1 Continuous Integration

For CI, we introduced the use of self-runners which is configured to Fedora version 34, to ensure that testing and assurance are performed on our selected choice of platform. Another reason for using self-runners instead of the runners deployed by the teaching team is because of the limitations imposed by the teaching team, as it does not match our testing framework capabilities.

Our automated testing consists of building the project from a clean state, running unit, integration, and system testing of various test suites. Upon completion of the test, our team will have the confidence that the new code addition is able to pass regression test and execute.

Figure 41 Unit and Integration Test CI

Besides unit and integration test, our team had engaged in using AutoTester to run every single source and query pair to ensure that our system can run fully. The CI will indicate the number of queries, together with the count of pass and failures from the output file. If required, the log files and output xml files are available to download for debugging. Refer to the figure below.

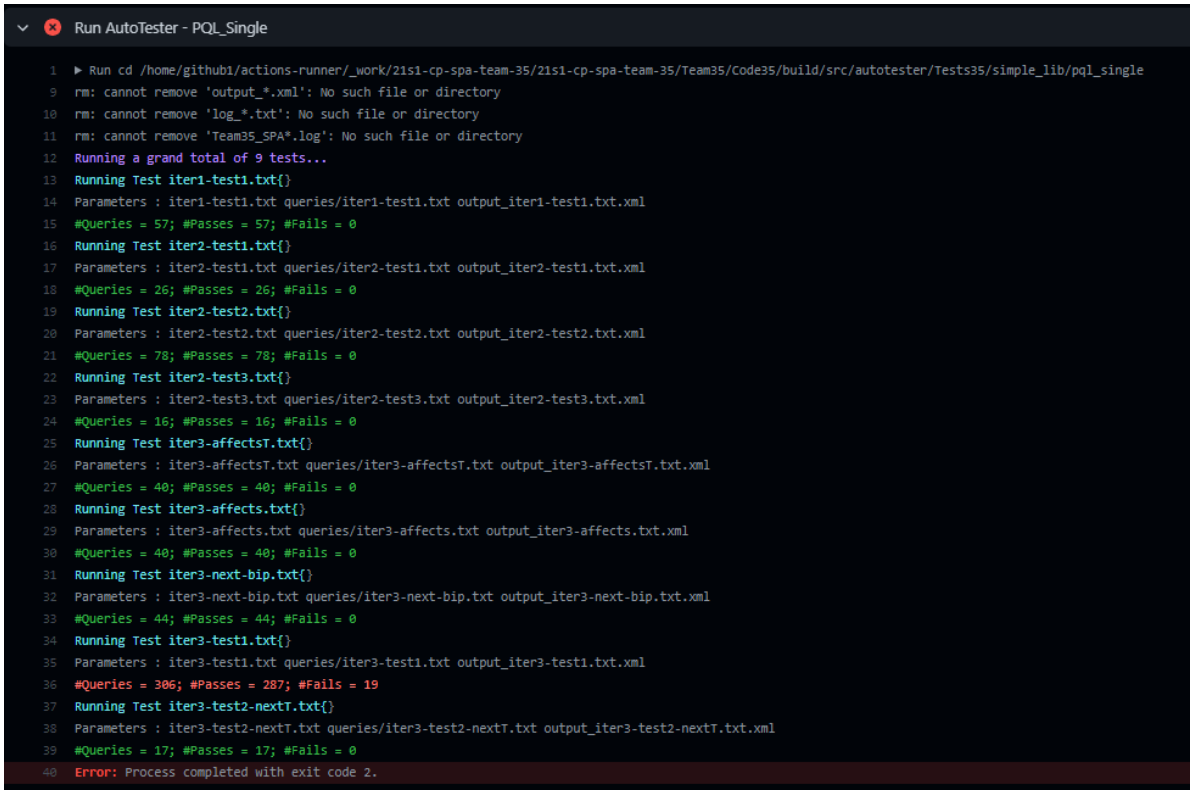


Figure 42 AutoTester Script CI

Running AutoTester manually on many source and query files will be a time-consuming chore and unverifiable by team members. With CI, we can solve this issue and perform such checks quickly. Using the figure above, we can check the number of passed and failed test cases immediately based on the coloured output.

5.1.2 System test automated creation

For the SIMPLE program for our stress testing, it would not be feasible nor accurate to create a program spanning more than 500 lines while including complex assignment expressions or multi-level nesting. Therefore, we utilized a python script to automate this SIMPLE source generation. This also ensures that it is easy to adjust the requirements of the source program since we can easily tweak the input settings in the python script generator.

When it comes to the queries, using a python script to generate the EPs would not only save a significant amount of time, but would also ensure that we do not miss out on any of the EP cases due to human error. In addition, changes to the queries would also be relatively straightforward through tweaking of the python script.

5.2 Tracking of defects

Our team utilizes Github Issues to keep track of bugs and fixes. Github Issues features milestones, tags and filters functionality, which our team is able to fully use to our advantage. The tags allow us to know which area the bug belongs to, as well as the urgency to the bugs. We can also ensure that all issues are cleared before a

crucial sprint milestone and backlogs. In addition, issue threads are available for our team to discuss potential bug fixes and enhancements, as well as knowing who is assigned to fix the bug.

5.3 Defect Resolution Time

If the error was caused by an old bug, the developer will open a new issue and assign it to the team accountable. The developer will also message everyone in a text channel to ensure that everyone is aware of the issue, and our team aims to resolve any potential urgent bug reports within 50 hours.

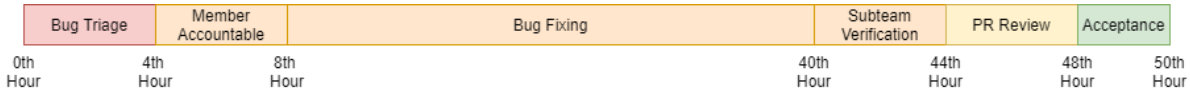


Figure 43 Bug Report Timeline

Step	Description
Bug Triage	Check if the bug is valid, reproducible, and priority for fixing.
Member Accountable	If the bug is deemed urgent, the member who created the function, or knows the function the best, will attempt to fix the issue.
Bug Fixing	The member or sub team who will fix the bug.
Subteam verification	As our team works in subgroups for variables components, each subteam must review the change and check for any potential bug which might arise from the change.
PR review	A pull request will be created and the bug reporter will have to review the changes. Everyone in the team are also encouraged to review the code.
Acceptance	The changes will be merged into master branch.

Table 18 Bug Report Timeline Description

5.4 Test Planning

Based on earlier test strategy discussions (iteration 1 & 2) our team focused on the following pointers for testing.

1. Earlier Test Strategy discussion

From the issues and bugs that we had encountered during the Iteration 1 phase, we noticed that most of our bugs arise from missing unit test cases to test edge cases. This can also be inferred as unit testing are performed by the developer themselves,

so there might not have considered the case during implementation and hence left them out during unit testing.

For Iteration 3, our team created test discussions earlier and as a team, discussed about testing during the API discovery and discussion phase, so that everyone's inputs can be considered, analysed and the developer will be aware of any pitfall and edge cases during development.

2. Performance Testing

To make full use of continuous integration to test our system, our team believes in executing extensive and comprehensive tests. This is also to take into account the requirements of Advance SPA where queries are evaluated in a quick and time limited manner. As such, our team will utilize auto generation of SIMPLE code and queries to generate large SIMPLE code base and queries that will cover most of the edge cases.

Another way of making use of automation is to create cross product checks on tuples and the various permutations of such that, with and pattern clauses, to ensure a consist output time and result.

3. Team review for changes to any test cases

For Iteration 3, our team decided to take a stricter approach when changes are made to testcases. We came to this conclusion as anyone can change test case to "make it work", so our team's reliance on comprehensive test framework might be misplaced. As such, we require most of the team's approval of a testing change before allowing changes to be made.

Our strategy for testing is to make test comprehensive and checkable by everyone in the team. We believe we can achieve this by making full use of continuous integration.

6. Coding standards

Early in the requirements analysis phase, we established a common code style guide along with various workflows. Our team chose Google's C++ Coding Standard. We understand that style guides are opinionated and that certain guidelines can be controversial, hence we referred to this guide mainly for standardization for naming and other stylistic choices rather than religiously following implementation specific opinions (e.g. Google C++ discourages use of exceptions).

Stylistically, we made one key deviation from the Google C++ standard:

```
int* pointer; //pointer declaration according to Google C++ Standard
int* pointer; //pointer declaration followed by Team 35
```

This follows **Bjarne Stroustrup's coding style** for the declaration of pointers where the asterisk is together with the type before a space. Since our team members are more familiar with type-based languages like Java, it would be easier for the team to comprehend the statement as an integer-type pointer.

This also prevents confusion in the syntax for dereferencing a pointer. We have followed Google's standard for dereferencing pointers. However, we have to be aware of a drawback which may trip new users.

```
*pointer = 1;
int* a, b; //variable a is a pointer, whereas variable b is an int
```

This won't be a problem since Google's guideline discourages multiple declarations with pointers or reference decorations in the same line.

7. Correspondence of the abstract API with the relevant C++ classes

Establishing the style guide early meant that the choice of function and class names used in Abstract API automatically followed this style, eliminating the need for any mapping between the terms used for Abstract API and the actual names used for our C++ classes. In places where there were complicated collections of different standard types, we used structs/classes to implement the abstract datatypes. However, if the Abstract datatype could have been implemented via native data-structures (e.g., pairs, vectors), we simply utilized them. In summary, there was a 1-1 correspondence between our Abstract API and our Concrete API implementations.

Part 3 – Conclusion

8. Reflections

This section chronicles the positive and negative experiences that we had during the project iterations, as well as our learnings. The reflection touches on four pillars – Communication within the team, project management, development practices, as well as the bug finding process.

Firstly, regarding communication, our key learning was to communicate clearly and often. This helped keep the rest of the team on board with each member's progress as well as to resolve issues early on during the development phase of the project. In addition, this helped with resolving changes early to prevent delays in project completion. To achieve this, we held weekly team meetings, and made use of platforms like Discord to engage in active communication.

Secondly, for project management, we had a positive experience because we made the effort to schedule and follow mini-iterations. This allowed us to compartmentalize the overall complexity of the project into more manageable chunks, which was especially crucial since the development team was still gaining familiarity with the project requirements, internal workflows, and the tools/programming language. This meant that we were able to make progress more confidently on the project from the very beginning, thereby achieving a good pace of continuous delivery. Looking back, in iteration 1 there were issues in the way we managed the project. Due to unfamiliarity with projecting the complexity of the tasks and analyzing the subcomponent functionalities, we ended up having uneven workload allocations for each sprint, resulting in backlogs within each sprint that had to be stacked onto the deliverables for the future sprints. However, after overcoming these teething issues, we experienced a more mature workflow for iteration 2 & 3, which was very helpful given iteration 3 had an even tighter time crunch, requiring us to really optimize our processes.

Thirdly, we ensured that our requirements planning translated well to the development process by planning the responsibilities of components and going through the data flow of the SPA by making sequence diagrams. This was helpful in surfacing doubts and ensuring a smoother coding process. Throughout the individual development process, we found it beneficial to actively make use of the features built into GitHub – we made heavy use of issue tracking, which was helpful for ensuring ownership and accountability for task assignments. On this point, we found that since code reviews were essential for us to have visibility on the quality of features developed by our teammates, we ensured that we kept our pull requests small and properly documented. This also allowed us to involve multiple members of the team (including those across sub-teams) in the review process. We also engaged in pair programming sessions between members in a sub-team, as well as during integration across components to ensure that we were on the same page with each other and to avoid information loss.

Lastly, for the bug management process, something we did well was to create bug reports whenever bugs were discovered. This helped to coordinate the reconciliation process efficiently, and as mentioned earlier, we achieved this through GitHub issues. However, when reflecting on the way we handled software testing, one issue we faced was that due to time constraints, our unit testing was not as comprehensive as expected. This was not ideal, as it meant that bugs in individual parts only surfaced closer to the deadline when system testing was being implemented. Moving forward, our team agrees that we should create test cases (at all levels) earlier in the iteration cycle by more closely adhering to a Test-Driven Development (TDD) approach as taught in class to introduce fewer regressions.

9. Assumptions Made

Functionality	Assumption	Remarks, if any
Query Parsing	1:Duplicate synonym declaration is treated as semantically invalid 2:‘`such that`’ is treated as one keyword during parsing, i.e. it must have exactly one whitespace in between.	1:note that we return none even for BOOLEAN queries for this specific case, as its not meaningful to return FALSE.
Constant Value	Any “INTEGER” as described in Syntax Reference will have a maximum limit of 2,147,483,647.	This is following the reasoning that the lexical token was coined “integer” where assumptions could be made as the maximum integer allowed by C++, rather than the mathematical term.

Table 19 Project Assumptions

10. Glossary of Terms

Term	Meaning
Statement	Statement is an implemented object defined in Entities below. Statements are delineated by these 3 characters: { ; } Thus, in addition to the Abstract Syntax Grammar provided, Statement can include Else and might not conform to its statement number.
Synonym	Synonym is an implemented object as defined in the domain of SPA, which seeks to model the declarations made in a PQL query. Similarly, our Synonym object has two attributes; a name as well as a type corresponding to a design entity (e.g. variable, constant, assign etc).
Clause	Clause is a structure that could either be of type ‘such that’ or ‘pattern’. For a given PQL query, we extract the essential information from (possibly multiple) clauses that are provided and internally represent them in the Query Processor by creating the corresponding clause objects.
Group	Group is an object that can be thought of as a list of Clauses, and some associated attributes. Conceptually, grouping clauses together allows the Query Processor to evaluate those clauses together to ensure correctness of the query.

Table 20 Glossary

11. Appendix

A. PKB Abstract API

API	Description
PKB	
14. VOID PopulateDataStructures(DELIVERABLE d)	Receives a Deliverable object, containing all entities and relationships between those entities, and stores that information within curated data structures within the PKB.
15. VOID PopulateRelationship(REL_MAP map, REL_TYPE rs_type)	Receives a relationship map and a relationship type and populates the data structures in the PKB with that relationship.
16. RELATIONSHIP_MAP GetRelationshipMap(REL_TYPE rs_type)	Receives a relationship type and returns a map of entity names to other entities in a relationship with the specified entity.
17. ENTITY_LIST GetRelationship(REL_TYPE rs_type, ENTITY_NAME entity)	Receives a relationship type and an entity name (Statement number, variable name, procedure name) and returns a list of all entities that possess the relationship with the specified entity. Returns an empty list if there is none.
18. ENTITY_LIST GetRelationshipByType(REL_TYPE rs_type, ENTITY_TYPE entity_type)	Receives a relationship type and an entity type and returns a list of entities of the specified entity type that possess that relationship. Returns an empty list if there is none.
19. ENTITY_PAIR_LIST GetRelationshipByTypes(REL_TYPE rs_type, ENTITY_TYPE type_1, ENTITY_TYPE type_2)	Receives a relationship type and two entity types and returns all pairs of entities of the specified types between which the specified relationship holds. Returns an empty list if there is none.
20. ENTITY_LIST GetDesignEntities (ENTITY_TYPE entity_type)	Receives an entity type and returns a list of all entities belonging to that type. Returns an empty list if there is none.
21. ENTITY_LIST GetPatternEntities(ENTITY_TYPE entity_type, ENTITY_NAME var_or_stmt)	Receives an entity type (While, Assign or If) and a variable name or statement number and returns matching pattern entities of the specified type.
22. ENTITY_LIST GetEntitiesWithAttributeValue(ENTITY_TYPE	Receives an entity type, attribute type and attribute value and returns a vector of entities of that specified type with that attribute value.

entity_type, ATTRIBUTE_TYPE attribute_type, ATTRIBUTE_VALUE attribute_value)	
23. ENTITY_PAIR_LIST GetEntitiesWithMatching Attributes(ENTITY_AND_ ATTRIBUTE_TYPE type_1, ENTITY_AND_ATTRIBU TE_TYPE type_2)	Receives two Entity/Attribute type combinations and returns a list of all entity pairs of those types that have matching attribute values.
24. BOOL HasRelationship(REL_TY PE rs_type)	Receives a relationship type and returns a Boolean of whether the relationship exists in the source SIMPLE code or not.
25. BOOL HasRelationship(REL_TY PE rs_type, ENTITY_NAME entity)	Receives a relationship type and an entity name and returns a Boolean of whether the specified entity has the relationship.
26. BOOL HasRelationship(REL_TY PE rs_type, ENTITY_TYPE type_1, ENTITY_TYPE type_2)	Receives a relationship type and two entity types and returns a Boolean of whether the relationship exists between any two entities of the specified types in the source SIMPLE code or not.
27. BOOL HasRelationship(REL_TY PE rs_type, ENTITY_NAME entity_1, ENTITY_NAME entity_2)	Receives a relationship type and two strings identifying entities and returns a Boolean of whether a relationship of the specified type between two entities with those identifiers exists in the source SIMPLE code or not.

B. Source Processor Abstract API

While the abstract API for PKB is used for communication between components (Source Processor, Query Processor), this API is mainly for communication between subcomponents of the Source Processor and planning.

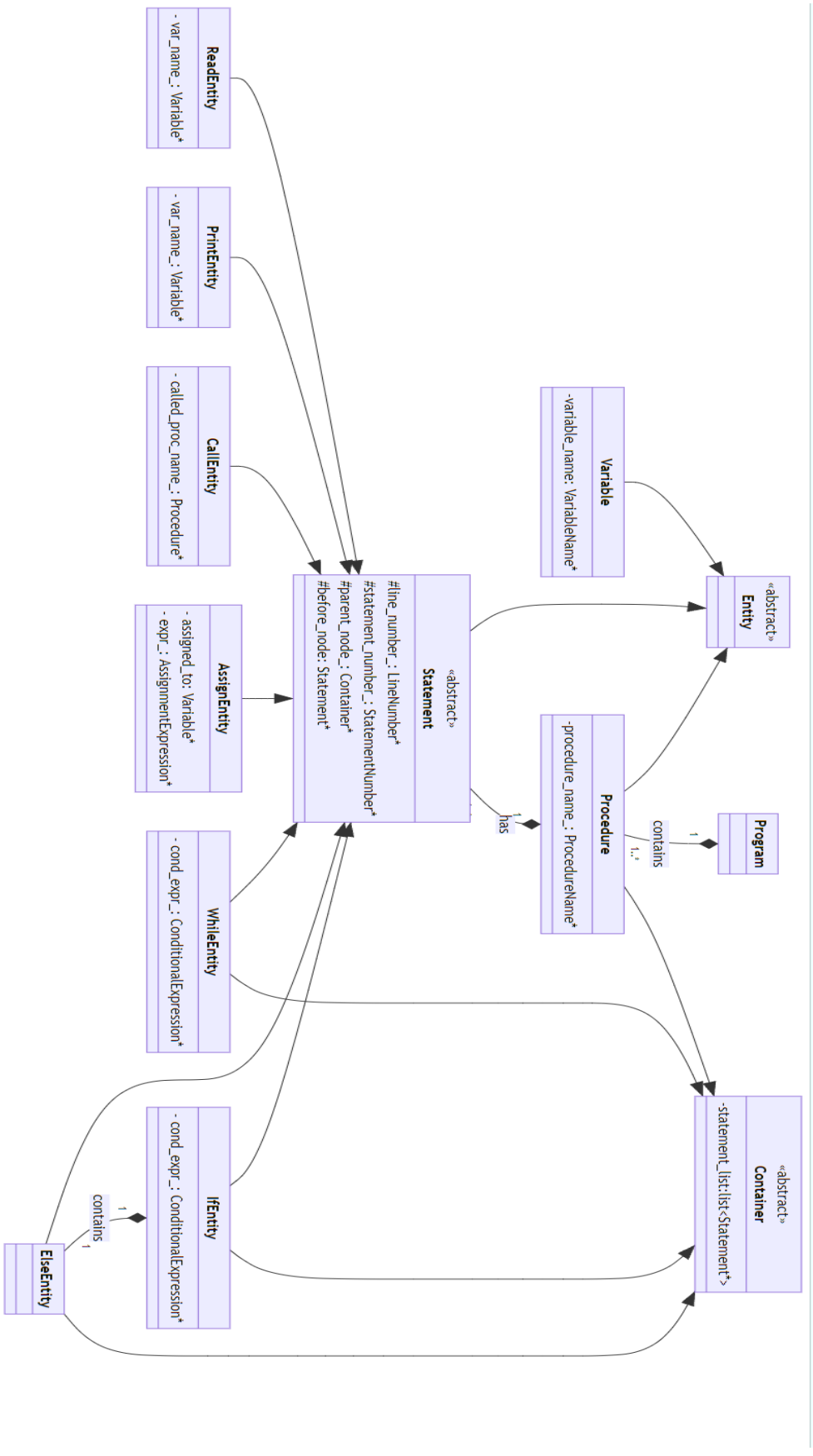
API	Description
SourceProcessor **Overview**: SourceProcessor manages the data flow between the Parser, DesignExtractor and the PKB. It holds the necessary data structures(Deliverables) for preprocessing of source code and populates the PKB when the preprocessing is done.	
1. PKB* ProcessSourceFile(STRING sourceFilePath)	Processes the source file at the path given, and returns a PKB pointer to the PKB that is populated with the necessary data.
2. VOID Terminate(STRING message)	Terminates Parser execution and logger, and exits program.
Parser **Overview**: Parser is a facade that takes in the source program from the SourceProcessor, checks the syntax, creates the necessary data structures and sends these Deliverables back.	
3. VOID Parse(STRING sourceFilePath)	Parses the source file at the path given, by calling the PSubsystem.
4. DELIVERABLE* GetDeliverables()	Wraps the necessary data structures in a Deliverable object and sends the pointer of the Deliverable to the SourceProcessor.
PSubsystem **Overview**: PSubsystem contains data structures that accumulate(EntityTables, RelationshipTables, Helper Stacks that keep track of tokens and determine how the Deliverables are being modified). It receives a single Source Statement from the Parser and initiates a pipeline for processing it: the statement is tokenized then its syntax is validated, entity objects are created and relationships are added.	
5. VOID InitDataStructures()	Initializes the data structures for the Deliverable and helper stacks.
6. VOID ProcessStatement(STRING stmt)	Tokenizes the statement, validates its concrete syntax and creates Entities and relationships that will then be added to the Deliverable.

7. DELIVERABLE* GetDeliverables()	Wraps the necessary data structures in a Deliverable object and sends the Deliverable to the Parser.
Tokenizer **Overview**: Tokenizer is responsible for converting the string format of a Source Statement into a list of lexical tokens, as defined in the Concrete Grammar Syntax.	
8. TOKEN_LIST CreateTokens(STRING stmt)	Creates tokens out of the statement according to the Syntax.
SyntaxValidator **Overview**: SyntaxValidator checks whether the tokenized statement follows the Concrete Grammar Syntax.	
9. BOOL ValidateSyntax(TOKEN_LIST tokens)	Validates the syntax of the vector of tokens.
EntityFactory **Overview**: EntityFactory is responsible for creating the Entity objects from the tokenized statement provided. Entity is an Abstract class.	
10. ENTITY* CreateEntities(TOKEN_LIST tokens)	Creates Entities based on the statement represented by the vector of tokens passed in.
DesignExtractor **Overview**: Extracts program design abstractions that cannot be extracted in the one pass of parsing of the source code, namely the transitive relationships.	
11. VOID ExtractDesignAbstractions()	Extracts program design abstractions from the data available after parsing and populates the relevant tables.

C. Query Processor Abstract API

API	Description
QueryExtractor **Overview**: QueryExtractor is a façade that calls subcomponents to tokenize, parse and validate a query, before applying some optimizations to the extracted data.	
1. VOID ExtractQuery(STRING query)	Processes the query string, modifying the query objects within the QueryExtractor object in-place with the extracted information.
2. VECTOR_OF_PAIRS<SYNONYM*, ATTRIBUTE> GetTargetSynAttrPairs()	Getter method for retrieving the target synonyms & the corresponding Attribute specified in the user query.
3. VECTOR<GROUP*> GetGroupsList()	Getter method for retrieving the references to all the Group objects created from 'grouping' related clauses in the user query.
QueryEvaluator **Overview**: QueryEvaluator is responsible for processing the information extracted from the queries and to call the respective clause type evaluator's to populate the table.	
4. VOID QueryEvaluator(PKB* pkb);	Initialiser for the QueryEvaluator instance, which takes in a pointer to a PKB object.
5. UnformattedQueryResult EvaluateQuery(VECTOR<Group* > groups)	Evaluates the query, returning the raw table results within a wrapper class called UnformattedQueryResult.
QueryProjector **Overview**: QueryProjector receives raw results of query evaluation and helps with the formatting of the output of the query to match the requirements of the Autotester.	
6. VECTOR<STRING> FormatQuery(UnformattedQuery Result unformatted_result)	Receives raw results of query evaluation as a custom wrapper object of type UnformattedQueryResult, and formats them before projecting the list results to the caller (TestWrapper).

D. Appendix – Class Diagram (Entity)



E. Iter 2 Extension

For our extension proposals to SPA, we suggest two potential extensions. Namely, these are the Dominates/Postdominates relationship as well as McCabe Cyclomatic Complexity analysis.

E.1 Dominates/Postdominates Relationship

The Dominates relationship is a relationship between statements. It is defined as follows: given two statements, s1 and s2, s1 dominates s2 if all control-flow paths that go through s2 go through s1.

The Dominates relationship can be useful in several cases. Determining that Dominates(s1, s2) holds true ensures that in all cases, s1 will execute before s2, which can be useful in the case where the execution of s1 is a prerequisite to s2 executing properly. Below we outline a specific example where the Dominates relationship can be used.

```
procedure dominates {
    x = 0;
    if (x == 0) then {
        y = 0;
    } else {
        x = 0;
    }
    print y;
}
```

In this example, the initialization of y (statement 3) is in the if-block of an if-statement. This means that it is possible that y does not get initialized with any value. However, y is printed (statement 5) after the if-statement. If the else-block is executed instead, the print statement would have undefined behavior.

To check for possible bugs arising from this, the Dominates relationship can be called. Dominates(3, 5) is false in this relationship, indicating that it is possible statement 4 is executed without statement 3. Note that the Next relationship already defined in SPA does not suffice to prevent this bug. To illustrate this, Next(3, 5) holds true. This is because while Next ensures that there exists a path between two statements, s1 and s2, it does not exclude the possibility of other paths leading to s2.

Using PQL, candidates for these bug-prone areas can be found easily using a combination of Modifies, Uses and Dominates.

```
// Query 1:
statement s1, s2; variable v;
Select <s1, s2> such that Modifies(s1, v) such that Uses(s2, v) such that Next(s1, s2)
// Query 2:
statement s1, s2; variable v;
Select <s1, s2> such that Modifies(s1, v) such that Uses(s2, v) such that Dominates(s1,
s2)
```

In the above queries, query 1 returns a tuple of all statement pairs in which the first statement modifies a variable and the second statement subsequently uses that variable. Query 2 returns a subset of these tuples in which the first statement always executes before the second statement. Listing the tuples returned in query 1 but not in query 2 therefore shows potential bugs. For instance, when run against the sample procedure above, the result of query 1 is (<1, 2>, <3, 5>) and that of query 2 is (<1, 2>), indicating that the control path between statements 3 and 5 is potentially problematic.

Postdominates

The Postdominates relationship is the opposite of this relationship. It is defined as follows: given two statements s1 and s2, s2 postdominates s1 if all control-flow paths through s1 go through s2. Note that Postdominate is not simply the inverse mapping of Dominates, as Dominates is based on paths entering s2 while Postdominates is based on paths exiting s1.

The Postdominates relationship can be useful in several cases. Determining that Postdominates(s2, s1) holds true ensures that in all cases in which s1 is executed, s2 executes afterward. This can be useful in cases in which s2 needs to be executed after s1. Below we outline a specific example where the Postdominates relationship can be used.

#	Statement
	procedure postdominates {
1	x = 0;
2	while (x < 5) {
3	if (x == 0) then {
4	x = x + 1;
	} else {
5	x = x – 1;
	}
	}
	}

In this example, the initialization of x (statement 1) is before a while statement. The variable is then incremented (statement 4) within an if statement. Since

Postdominates(4, 1) is false, this means that there is potential for an infinite loop here, as it is possible for x to never be incremented.

Other Benefits

In addition to the debugging use cases outlined above, the following are reasons the Dominates/Postdominates relationships may prove useful to developers.

1. Identify critical sections for testing/assertion

If a certain statement dominates or postdominates many other statements, this is an indicator that the statement may be integral to the successful execution of the program. The developer may therefore choose to write more test cases to ensure the correctness of code in these sections. Assertions might also be useful to ensure these sections of code work according to expected behavior (assuming SIMPLE allows for assertions).

2. Prevent Single Point of Failure (SPOF)

If there exists a statement that postdominates all statements before it and dominates all statements after it, this statement can be considered a single point of failure for the entire program. If this statement does not execute correctly, the entire program is likely to crash. Therefore, to avoid such a scenario from happening, it may be desirable to refactor the code to reduce the instances of such SPOF statements.

3. Refactoring considerations

Dominates and Postdominates have particular relevance in the refactoring of code. For example, if one statement postdominates all statements before it and a subsequent statement dominates all statements after it, the region of code between the two statements (inclusive) may be abstracted out into a separate procedure.

Possible Implementation Challenges

1. Correctness of Control Flow Graph (CFG)

The implementation of Dominates and Postdominates both rely on the creation of the complete CFG. An incomplete CFG will not suffice as it is possible that there are missing edges to/from a specific node that will result in false positives for both Dominates and Postdominates. This means that the CFG that is passed to the depth-first-search extractors must be complete and accurate.

2. Complexity of Dominates algorithm

The naïve algorithm to solve the Dominates algorithm is as follows: beginning at a root node (the starting point of the program), first conduct a depth-first search to find all reachable nodes through the execution of the program. Subsequently, for each node in the program, remove the node and conduct depth-first search from the root node once again. All nodes that were reached in the first run but not the second are dominated by the specified node. Unfortunately, this solution runs in quadratic time with respect to the number of nodes.

Fortunately, there exists an O(m log n) solution, where m is the number of edges and n is the number of nodes. This algorithm is outlined here, in “A Fast Algorithm for Finding Dominators in a Flowgraph” by Lengauer and Tarjan. However, this solution is complicated and requires the creation of several auxiliary graphs and relationships.

Postdominates poses the same implementation challenges as Dominates as it is simply in the reverse order.

Effect on SPA Components

The SPA Components that will be affected to support these extensions range across the entire SPA. These include the DesignExtractor, the PKB, the QueryExtractor and QueryEvaluator.

1. DesignExtractor

The DesignExtractor must be modified to extract the Dominates and Postdominates relationships. This would involve implementing a depth-first-search from the target node backwards to the start node (for both Dominates and Postdominates). This depth-first-search algorithm must ensure that all paths traversed eventually reach the start node. This suffices to ensure the Dominates/Postdominates relationships hold.

2. Deliverable

The Deliverable object passed to the PKB must be refactored to include Dominates and Postdominates (forward and reverse) relationship hashmaps. These hashmaps would map from entity to list of entities, similar to other existing hashmaps.

3. PKB

The PKB must be extended in order to add the Dominates and Postdominates relationships. The different new relationships (Dominates, DominatedBy, Postdominates, PostdominatedBy) must be added to the enum class of relationship types. The PopulateRelationship method will also have to be called for each of these in PopulateDataStructures. Otherwise, the functionality of the PKB remains the same.

4. QueryEvaluator

The QueryEvaluator must be refactored to understand that the new relationship types may be found in the PKB. This involves minimal refactoring (addition of cases to a switch case).

5. QueryExtractor

The QueryExtractor must be refactored to recognise Dominates and Postdominates as relationship types.

E.2 McCabe Cyclomatic Complexity analysis

McCabe Cyclomatic Complexity is a quantitative metric that measures the complexity of a program. Specifically, it measures the number of independent paths within the CFG of a program. Each path refers to a control flow execution of statements, and for a path to be independent of other paths, it must have at least one edge not traversed in any other path. The higher the complexity, the larger the number of potential execution flows within the program and therefore the more prone the program is to bugs.
Cyclomatic complexity can be measured for a program as a whole or for specified sections of the program (e.g. procedures), for instances in which only a region of code is under scrutiny.

Benefits

1. Allows comparison between analogous programs

Given two approaches that accomplish the same task, McCabe Cyclomatic Complexity offers a way to compare the code complexity of each approach. Along with the developer's understanding of other factors (e.g. time/space complexity), this measure may therefore be helpful in choosing one over the other.

2. Testing can be more systematic

As cyclomatic complexity is equal to the number of independent paths in a program, it also corresponds to the upper bound to the number of test cases needed in order to obtain full branch coverage. It is also a lower bound for the number of test cases needed in order to obtain full path coverage, and serve as a preliminary measure of whether full path coverage has been achieved or not.

3. Assist with development

With every new control statement introduced, the cyclomatic complexity of the program will increase. The developer can therefore empirically decide on a complexity that is too high, based on the relationship between his code quality and cyclomatic complexity. If a specific portion of code has cyclomatic complexity over this threshold, it might be best to refactor the code to abstract out different subcomponents.

Effect on SPA Components and Potential Implementation Challenges

The SPA Components that will be affected lie primarily in the Query Extractor and Evaluator sections of the SPA. Cyclomatic analysis is based on CFG analysis. As the CFG has already been created and stored in the PKB, components that execute prior to and during the PKB population need not be refactored. However, it may be non-trivial to calculate the complexity for specific subsections of code, as this would require separate analysis into only specific parts of the CFG.

F. Stress testing automation

For the generation of the stress test, our team utilized a python script to generate complex expressions with random levels of nesting.

```
import random

RANDOM_EXPRESSION_VALUE = 1
RANDOM_EXPRESSION_MIN_VALUE = 1
RANDOM_EXPRESSION_MAX_VALUE = 30
RANDOM_CONDITIONAL_VALUE = 1
MAX_NESTING_LEVEL = 4

number_of_statements = 100
variable_list = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"]

    #print random statement
def printRandStmt(level_of_nesting):
    if (random.randint(0, 2) == 0 and level_of_nesting < MAX_NESTING_LEVEL):
        return stmt_list[random.randint(0, 4)](level_of_nesting + 1)
    else:
        return stmt_list[random.randint(0, 2)](level_of_nesting + 1)
# types of statements
# print
# read
#assign
#while
#if
#call(?)
def printStmt(nesting_level):
    var = random.choice(variable_list)
    return lineIndentation(nesting_level) + "print " + var + ";\n"

def readStmt(nesting_level):
    var = random.choice(variable_list)
    return lineIndentation(nesting_level) + "read " + var + ";\n"

def assignStmt(nesting_level):
    lhs_variable = random.choice(variable_list)
    rhs_expression = random_expression()
    return lineIndentation(nesting_level) + lhs_variable + " = " + rhs_expression +
";\n"

def random_expression():
    return str(randomExpression(RANDOM_EXPRESSION_VALUE))

def whileStmt(nesting_level):
    whileStmtString = lineIndentation(nesting_level)
    whileStmtString += ("while (")
    whileStmtString += (randomConditional(RANDOM_CONDITIONAL_VALUE))
    whileStmtString += (") {\n")
    for i in range(random.randint(1, 10)):
        whileStmtString += lineIndentation(nesting_level + 1) +
printRandStmt(nesting_level + 1)
        whileStmtString += "\n"
    whileStmtString += ("}\n")
    return whileStmtString

def ifStmt(nesting_level):
    ifStmtString = lineIndentation(nesting_level)
    ifStmtString += ("if (")
    ifStmtString += (randomConditional(RANDOM_CONDITIONAL_VALUE))
    ifStmtString += (") then {\n")
```

```
57.     for i in range(random.randint(1, 10)):
58.         ifStmtString += lineIndentation(nesting_level + 1)
59.         ifStmtString += printRandStmt(nesting_level + 1) + "\n"
60.     ifStmtString += lineIndentation(nesting_level) + ("} else {") + "\n"
61.     for i in range(random.randint(1, 10)):
62.         ifStmtString += lineIndentation(nesting_level + 1)
63.         ifStmtString += printRandStmt(nesting_level + 1) + "\n"
64.     ifStmtString += lineIndentation(nesting_level) + "}"
65.     return ifStmtString
66.
67. class Expression:
68.     pass
69.
70. class Number(Expression):
71.     def __init__(self, num):
72.         self.num = num
73.
74.     def __str__(self):
75.         return str(self.num)
76.
77. class Variable(Expression):
78.     def __init__(self, var):
79.         self.var = var
80.
81.     def __str__(self):
82.         return str(self.var)
83.
84. class BinaryExpression(Expression):
85.     def __init__(self, left, op, right):
86.         self.left = left
87.         self.op = op
88.         self.right = right
89.
90.     def __str__(self):
91.         return str(self.left) + " " + self.op + " "  + str(self.right)
92.
93. class ParenthesizedExpression(Expression):
94.     def __init__(self, exp):
95.         self.exp = exp
96.
97.     def __str__(self):
98.         return "(" + str(self.exp) + ")"
99.
100.  def randomExpression(prob):
101.      p = random.random()
102.      if p > prob:
103.          if random.randint(0, 1) == 0:
104.              return Number(random.randint(RANDOM_EXPRESSION_MIN_VALUE,
    RANDOM_EXPRESSION_MAX_VALUE))
105.          else:
106.              randVar = random.choice(variable_list)
107.              return Variable(randVar)
108.      elif random.randint(0, 1) == 0:
109.          return ParenthesizedExpression(randomExpression(prob / 1.3))
110.      else:
111.          left = randomExpression(prob / 1.2)
112.          op = random.choice(["+", "-", "*", "/", "%"])
113.          right = randomExpression(prob / 1.2)
114.          return BinaryExpression(left, op, right)
115.
116.  def randomConditional(prob):
117.      if random.random() > prob:
118.          return randomRelExpr()
119.      else:
120.          case = random.randint(0, 2)
121.          if (case == 0):
```

```
122.             return "! (" + randomConditional(prob/1.2) + ")"
123.         elif (case == 1):
124.             return "(" + randomConditional(prob/1.6) + ") && (" +
    randomConditional(prob/1.6) + ")"
125.         else:
126.             return "(" + randomConditional(prob/1.6) + ") || (" +
    randomConditional(prob/1.6) + ")"
127.
128.  def randomRelExpr():
129.      lhs = random.choice(variable_list) if random.randint(0, 1) == 0 else
    str(random.randint(RANDOM_EXPRESSION_MIN_VALUE, RANDOM_EXPRESSION_MAX_VALUE))
130.      rhs = random.choice(variable_list) if random.randint(0, 1) == 0 else
    str(random.randint(RANDOM_EXPRESSION_MIN_VALUE, RANDOM_EXPRESSION_MAX_VALUE))
131.      op = random.choice(["==", "!=", ">", ">=", "<", "<="])
132.      return lhs + op + rhs
133.
134.  def lineIndentation(nesting_level):
135.      return_string = ""
136.      for i in range(nesting_level):
137.          return_string += " "
138.      return return_string
139.
140.  stmt_list = [printStmt, readStmt, assignStmt, whileStmt, ifStmt]
141.
142.  file = open("random_simple_prog.txt", 'w')
143.  for i in range(100):
144.      file.write(printRandStmt(0))
145.
146.  file.close()
147.
```

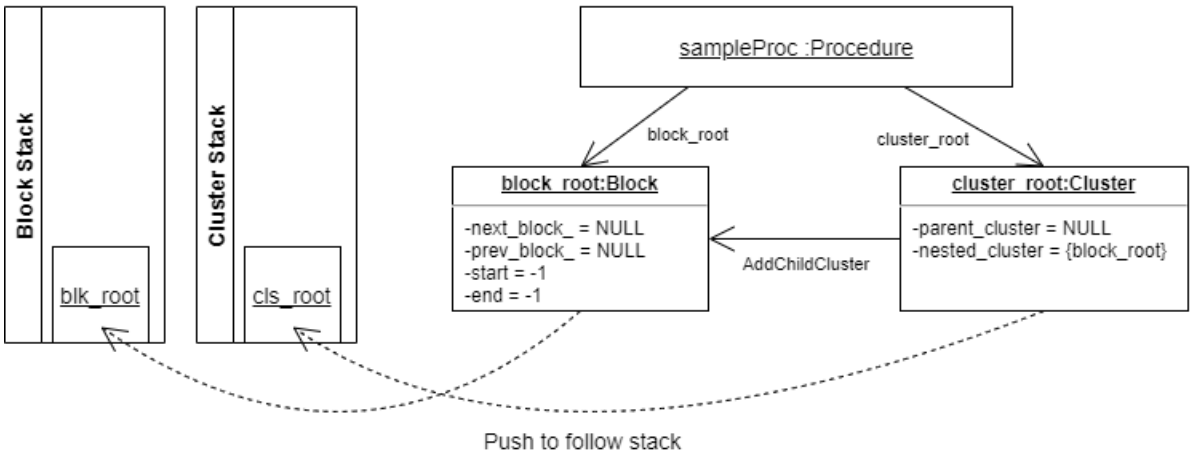
G. Automation of Query Generation

```
1.  import random
2.
3.  attrRef = ["pr1.procName", "c1.procName", "v1.varName", "r1.varName", "p1.varName",
    "const.value", "s1.stmt#", "r1.stmt#", "p1.stmt#", "c1.stmt#", "w1.stmt#",
    "if1.stmt#", "a1.stmt#"]
4.  declaration = "stmt s1, s2; assign a1, a2; call c1, c2; print p1, p2; read r1, r2;
    procedure pr1, pr2; while w1, w2; if if1, if2; prog_line pl1, pl2; variable v1, v2;
    constant const1, const2; "
5.  variable_list = ["x", "First", "noSuchVariable", ""]
6.  integer_list = ["-1", "0", "1", "100"]
7.
8.  count = 1
9.
10. def get_synonym_cross():
11.     total_query = ""
12.     for attr in attrRef:
13.         for attr2 in attrRef:
14.             global count
15.             query = str(count) + " - synonym cross\n"
16.             query += declaration + "\n"
17.             query += "Select s1 with " + attr + " = " + attr2 + "\n"
18.             query += "none\n"
19.             query += "5000\n"
20.             total_query += query
21.             count += 1
22.     return total_query
23.
24. def get_attrRef_variable_cross():
25.     total_query = ""
26.     for attr in attrRef:
27.         for var in variable_list:
28.             global count
29.             query = str(count) + " - attrRef-variable pair\n"
30.             query += declaration + "\n"
31.             if (random.randint(0, 1) == 0):
32.                 query += "Select s1 with " + attr + " = \"" + var + "\"\n"
33.             else:
34.                 query += "Select s1 with \"" + var + "\" = " + attr + "\n"
35.             query += "none\n"
36.             query += "5000\n"
37.             total_query += query
38.             count += 1
39.     return total_query
40.
41. def get_attrRef_integer_cross():
42.     total_query = ""
43.     for attr in attrRef:
44.         for int_value in integer_list:
45.             global count
46.             query = str(count) + " - attrRef-interger pair\n"
47.             query += declaration + "\n"
48.             if (random.randint(0, 1) == 0):
49.                 query += "Select s1 with " + attr + " = " + int_value + "\n"
50.             else:
51.                 query += "Select s1 with " + int_value + " = " + attr + "\n"
52.             query += "none\n"
53.             query += "5000\n"
54.             total_query += query
55.             count += 1
56.     return total_query
57.
58. def get_int_cross():
59.     total_query = ""
```

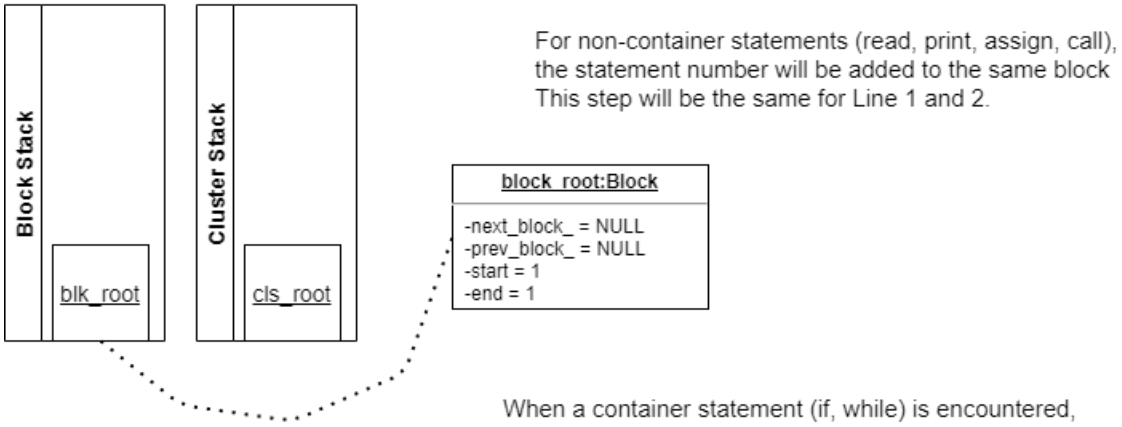
```
    for int1 in integer_list:
        for int2 in integer_list:
            global count
            query = str(count) + " - int cross\n"
            query += declaration + "\n"
            query += "Select BOOLEAN with " + int1 + " = " + int2 + "\n"
            query += "none\n"
            query += "5000\n"
            total_query += query
            count += 1
    return total_query

def get_var_cross():
    total_query = ""
    for var1 in variable_list:
        for var2 in variable_list:
            global count
            query = str(count) + " - variable cross\n"
            query += declaration + "\n"
            query += "Select BOOLEAN with \"" + var1 + "\" = \"" + var2 + "\"\n"
            query += "none\n"
            query += "5000\n"
            total_query += query
            count += 1
    return total_query

def get_var_int_cross():
    global count
    query = str(count) + " - variable-int pair\n"
    query += declaration + "\n"
    query += "Select BOOLEAN with " + "1" + " = " + "\"x\"" + "\n"
    query += "none\n"
    query += "5000\n"
    return query
    count += 1

file = open("with_clause_test_queries.txt", 'w')
file.write(get_synonym_cross())
file.write(get_attrRef_integer_cross())
file.write(get_attrRef_variable_cross())
 file.write(get_int_cross())
 file.write(get_var_cross())
 file.write(get_var_int_cross())

 file.close()

```

H. State Diagram for Blocks and Clusters creation

Line 0: procedure sampleProc {



Line 1: read x;



Line 3: while (a>=(1*(1/(10)))) {