CS3203 Software Engineering Project

Project Iteration 3

Team 03 (Consultation Hour: Tuesday 11am - 12pm)

Group - PKB
Jonathan Cheng | A0121749A | jonathan.cheng@u.nus.edu | 91852241
Lim Li | A0149289L | limli@u.nus.edu | 86023774
Tan Chee Kun, Thomas | A0168782M | thomastanck@u.nus.edu | 97860272
Group - Query Processor
Dominique Ng Wenyi | A0189690L | dominique@u.nus.edu | 98771356
Kenneth Fung Chen Yu | A0190198W | kenneth.fung@u.nus.edu | 96877924
Lee Shi Jie Shawn | A0190131W | shawnlsj@u.nus.edu | 81267220

Table of Contents

1 Scope

The table below outlines the capabilities of our SPA for Iterations 1, 2 and 3.

	Programs parsed	Relationships Answered	Clauses handled	Selected from query
Iteration 1	✔ SIMPLE syntax	✔ Follows/Follows* ✔ Parent/Parent* ✔ Uses ✔ Modifies	✔ Such that ✔ Pattern (assign)	✔ Statement ✔ Variable ✔ Procedure ✔ Constant ✔ Attribute
Iteration 2		✔ Calls/Calls* ✔ Next/Next*	✔ Pattern (with, ifs)	✔ Prog_line ✔ Tuple ✔ Boolean
Iteration 3		✔ Affects/Affects*	✔With	

As of Iteration 3, our Static Program Analyzer (SPA) has met all the basic and advanced requirements. In addition, we have also implemented **query optimization** in the SPA such that it is able to handle complex queries efficiently.

Our program is able to:

1.1 PKB

- Store and retrieve all design entities in the Advanced SPA requirements
- Store and retrieve Follows/Follows*, Parent/Parent*, Uses, Modifies, Calls/Calls* and Next relationships
- Compute Next* and Affects/Affects* relationships on-demand at runtime

1.2 Query Processor

- Validate syntax and answer queries written in Program Query Language (PQL) after SIMPLE source program has been processed
- Perform semantic validation and terminate query execution for semantically invalid PQL queries
- Perform query optimisation such that complex queries are handled efficiently
- The types of queries supported by our SPA correspond to the full PQL requirements, which we summarize below:
    - A valid PQL Query must consist of one **Select** clause, with any number and combination of design-entity relations that might be filtered by
        - Any number of **such that** clauses
        - Any number of **assign/while/if pattern** clauses

    - Any number of **with** clauses
- The valid entities to select in a **Select** clause are
    - Boolean
    - Tuples
    - Synonym
    - Synonym with attrName
- The valid relations to a query in a **such that** clause consists of any number of
    - Follows /*
    - Parent /*
    - Uses-(Statement / Procedure)
    - Modifies-(Statement / Procedure)
    - Calls /*
    - Next /*
    - Affects /*
    - NextBip /*
    - AffectsBip /*
- Valid **assign-pattern** clauses are
    - Partial match of an expression
    - Full match of an expression
    - Match to any expression ("_")
- Valid **while-pattern** or **if-pattern** clauses do a filter on the condition variables of the while or if statement by
    - A variable synonym
    - A variable literal
    - _ (all variable) match
- Valid **with** clauses consist of comparison between any two of the following
    - Synonym (only prog_line synonym)
    - Literal
    - Attribute Ref which is syn.attrName where attrName can be
        - value
        - procName
        - varName
        - stmt#
    - Any reference of type "STRING" must be compared to another "STRING" type, and "INTEGER" must be matched to "INTEGER" in the with clause
- Clauses can be separated by the keyword "and" provided that it is a similar type (such that / pattern / with) of clause

1.3 Special Achievements

Our team has implemented the extensions of NextBip, NextBip*, AffectsBip and AffectsBip* according to the definitions on the CS3203 wiki page.

2 Development Plan

This section describes how our team chose to allocate our time and manpower for the overall project and this iteration, along with our testing plan.

2.1 Project Structure

We decided to split the project into 2 main components - Front End & PKB and Query Processor - with 3 team members working on each main component.

Front End & PKB	Query Processor
● Thomas ● Jonathan ● Lim Li	● Shawn ● Dominique ● Kenneth

Team Roles:

- Team Lead x 1
- Tech Lead x 1
- Front End & PKB IC x 1
- Query Processor IC x 1
- Testing IC x 2
- Documentation IC x 1

We allocated two testing ICs from each component to ensure comprehensive testing. The tech lead had the strongest technical skill and provided assistance to all other members throughout the project development. Refer to Figure 2.1.1 below for our role and component allocation.

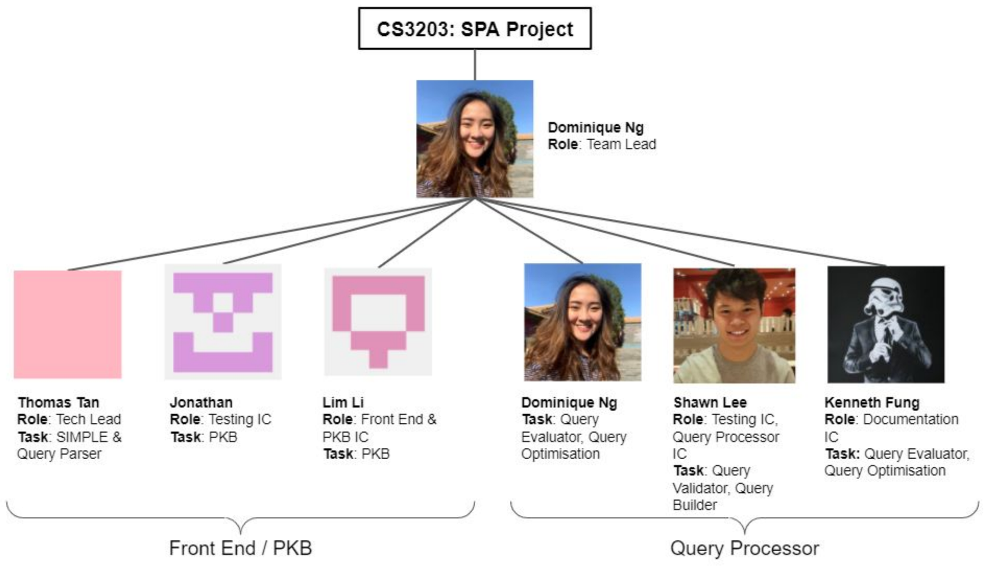

CS3203: SPA Project
Dominique Ng Role: Team Lead
Thomas Tan Role: Tech Lead Task: SIMPLE & Query Parser
Jonathan Role: Testing IC Task: PKB
Lim Li Role: Front End & PKB IC Task: PKB
Dominique Ng Task: Query Evaluator, Query Optimisation
Shawn Lee Role: Testing IC, Query Processor IC Task: Query Validator, Query Builder
Kenneth Fung Role: Documentation IC Task: Query Evaluator, Query Optimisation
Front End / PKB
Query Processor

Figure 2.1.1: CS3203 Project Group Structure

2.2 Project Timeline

In developing our SPA, we adopted the iterative software development process model, where the system was developed incrementally through repeated cycles. The iterative model allowed us to continuously review our performance and make improvements in subsequent cycles. We gradually learned more about the problem after each cycle and used our newfound knowledge to arrive at the solution.

More specifically, we adopted the breadth-first iteration model, where we worked on all components simultaneously to develop one functionality at a time. This way, we could conduct frequent integration and system testing at the end of each mini-iteration to ensure that we were meeting the requirements of the project. This also increased motivation as we were able to see actual results. Furthermore, we were able to identify bugs and design issues early and often, which allowed us to rectify our errors ahead of major deadlines.

We have divided the 3 main iterations into 9 mini-iterations. Refer to Figure 2.2.1 below for an illustration of our overall development timeframe.

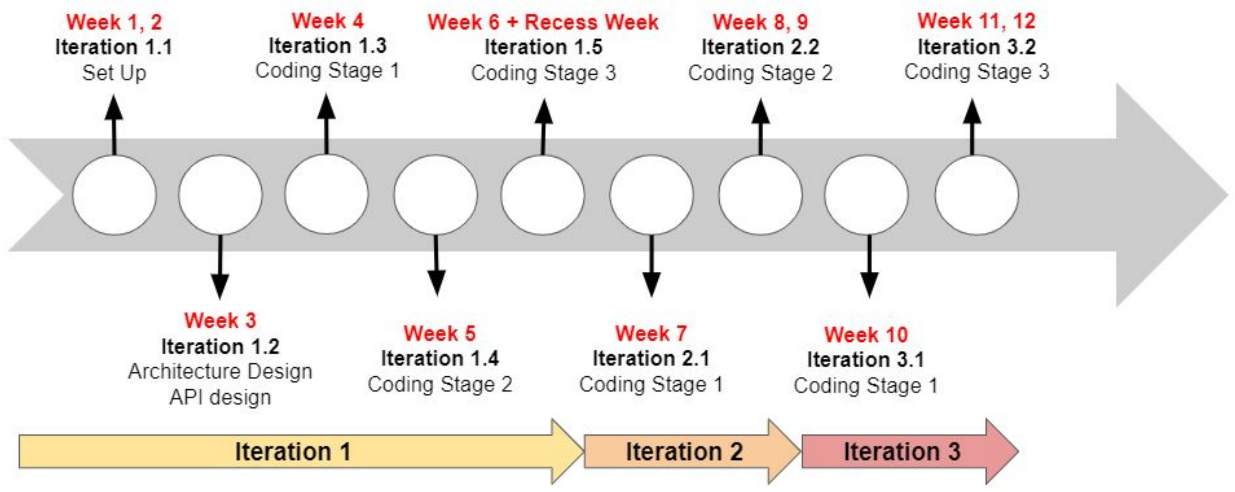


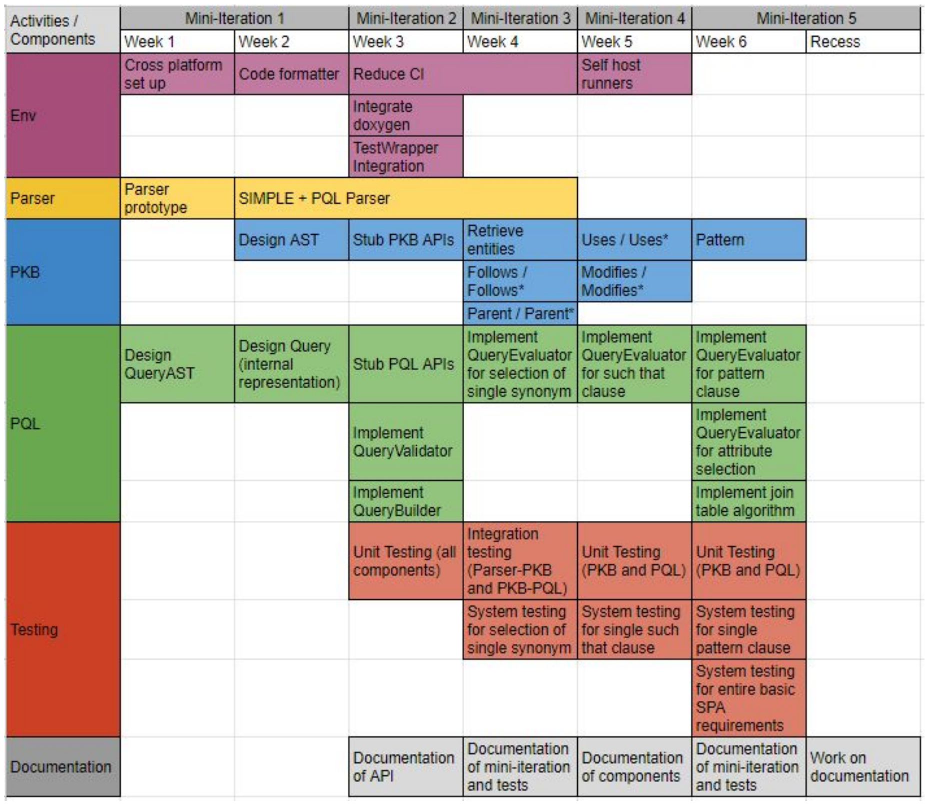
Figure 2.2.1: Overall Development Timeline

Each mini-iteration is conducted within 1-2 weeks and ends before our consultation with the TA.

2.3 Tasks and Activities

2.3.1 Overall Project Tasks

Iteration 1



Activities / Components	Mini-Iteration 1		Mini-Iteration 2	Mini-Iteration 3	Mini-Iteration 4	Mini-Iteration 5	
	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Recess
Env	Cross platform set up	Code formatter	Reduce CI		Self host runners		
			Integrate doxygen				
			TestWrapper Integration				
Parser	Parser prototype	SIMPLE + PQL Parser					
PKB		Design AST	Stub PKB APIs	Retrieve entities	Uses / Uses*	Pattern	
				Follows / Follows*	Modifies / Modifies*		
				Parent / Parent*			
PQL	Design QueryAST	Design Query (internal representation)	Stub PQL APIs	Implement QueryEvaluator for selection of single synonym	Implement QueryEvaluator for such that clause	Implement QueryEvaluator for pattern clause	
			Implement QueryValidator			Implement QueryEvaluator for attribute selection	
			Implement QueryBuilder			Implement join table algorithm	
Testing			Unit Testing (all components)	Integration testing (Parser-PKB and PKB-PQL)	Unit Testing (PKB and PQL)	Unit Testing (PKB and PQL)	
				System testing for selection of single synonym	System testing for single such that clause	System testing for single pattern clause	
						System testing for entire basic SPA requirements	
Documentation			Documentation of API	Documentation of mini-iteration and tests	Documentation of components	Documentation of mini-iteration and tests	Work on documentation

Iteration 2

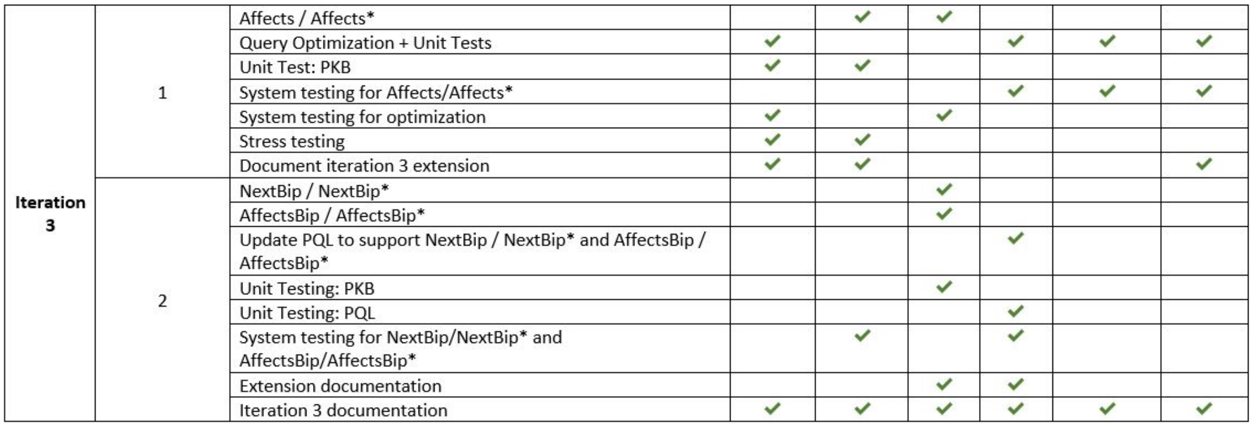
Tasks / Components	Mini-Iteration 1	Mini-Iteration 2	
	Week 7	Week 8	Week 9
PKB		Calls / Calls*	
		Next / Next*	
PQL	Refactor Query to support optimisation	Update QueryValidator for advanced SPA	QueryEvaluator for advanced SPA
		Update QueryBuilder for advanced SPA	
Testing	Regression testing for refactoring	Unit testing (PKB, QueryValidator and QueryBuilder)	System testing for Calls / Calls* and Next / Next*
Documentation	Document iteration 2 extension		Work on documentation

Iteration 3

Tasks / Components	Mini-Iteration 1		Mini-Iteration 2
	Week 10	Week 11	Week 12
PKB	Affects / Affects*		Extension (NextBip / NextBip* / AffectsBip / AffectsBip*)
PQL	Implement optimisation		Extension
Testing	Unit testing (PKB)	System Testing (Optimization)	Unit testing (PKB + PQL)
	System Testing (Affects / Affects*)	Stress Testing	System testing (extension)
Documentation	Document iteration 3 extension		Work on documentation

Overall Task Assignment

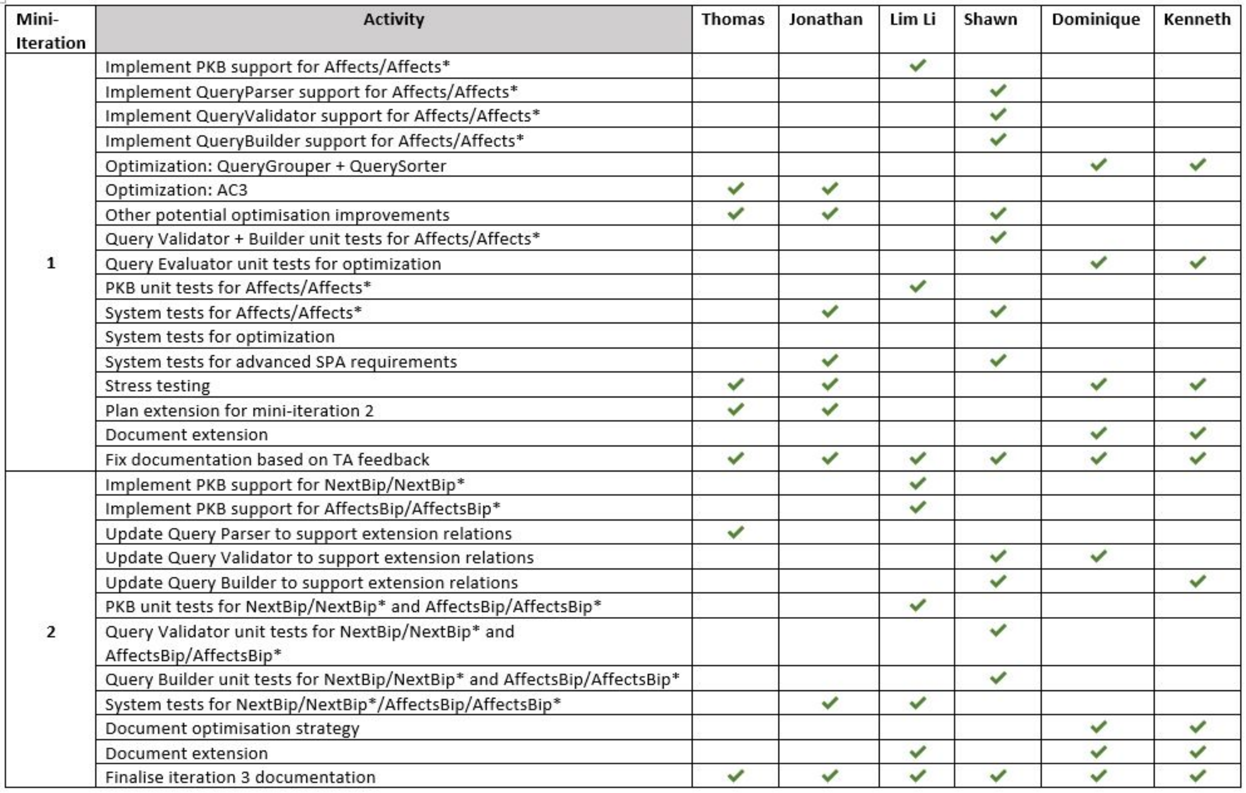
	Mini-Iteration	Task	Thomas	Jonathan	Lim Li	Shawn	Dominique	Kenneth
Iteration 1	1	Cross platform set up	✓					
		Parser prototype	✓					
		Design QueryAST	✓					
		Code formatter	✓					
		SIMPLE Parser + Unit Tests	✓					
		Design SIMPLE AST		✓	✓			
		Design Query				✓	✓	✓
	2	Set up continuous integration	✓					
		Integrate doxygen	✓					
		TestWrapper integration	✓	✓				
		PQL Parser + Unit Tests	✓			✓		
		Design PKB API		✓	✓			
		Design Query Processor API				✓		
		Implement QueryValidator + Unit Tests				✓	✓	✓
		Implement QueryBuilder + Unit Tests				✓		
		API documentation		✓	✓		✓	✓
	3	Retrieve design entities		✓	✓			
		Follows/Follows* relation		✓	✓			
		Parent/Parent* relation		✓	✓			
		Implement QueryEvaluator for selection of single synonym					✓	✓
		Integration testing: Parser – PKB		✓	✓			
		Integration testing: PKB – PQL				✓	✓	✓
		System testing for selection of single synonym				✓		
		Documentation of mini-iteration + tests	✓	✓	✓	✓	✓	✓
	4	Self-host runners	✓					
		Uses/Uses* relation		✓	✓			
		Modifies/Modifies* relation		✓	✓			
		Implement QueryEvaluator for such that clause					✓	✓
		Unit Testing: PKB		✓	✓			
		Unit Testing: PQL					✓	✓
		System testing for single such that clause	✓	✓		✓		
		Documentation of components	✓	✓	✓	✓	✓	✓
	5	Pattern matching		✓	✓			
		Implement QueryEvaluator for pattern clause					✓	✓
		Implement QueryEvaluator for attribute selection				✓	✓	✓
		Implement join table algorithm	✓				✓	✓
		Unit Testing: PKB		✓	✓			
		Unit Testing: PQL					✓	✓
		System testing for single pattern clause		✓		✓		
		System testing for entire basic SPA requirements		✓		✓		
		Documentation for iteration 1	✓	✓	✓	✓	✓	✓
Iteration 2	1	Refactor Query to support optimisation	✓					
		Regression testing for refactoring		✓				
		Document iteration 2 extension				✓		
	2	Calls / Calls* relation		✓	✓			
		Next / Next* relation		✓	✓			
		Update QueryValidator for advanced SPA				✓		
		Update QueryBuilder for advanced SPA	✓			✓		
		Update QueryEvaluator for advanced SPA					✓	✓
		Unit Testing: PKB		✓	✓			
		Unit Testing: QueryValidator				✓		
		Unit Testing: QueryBuilder				✓		
		System testing for Calls/Calls* and Next/Next*	✓	✓		✓	✓	✓
		Iteration 2 documentation	✓	✓	✓	✓	✓	✓



Iteration 3	1	Affects / Affects*		✓	✓			
		Query Optimization + Unit Tests	✓			✓	✓	✓
		Unit Test: PKB	✓	✓				
		System testing for Affects/Affects*				✓	✓	✓
		System testing for optimization	✓		✓			
		Stress testing	✓	✓				
		Document iteration 3 extension	✓	✓				✓
	2	NextBip / NextBip*			✓			
		AffectsBip / AffectsBip*			✓			
		Update PQL to support NextBip / NextBip* and AffectsBip / AffectsBip*				✓		
		Unit Testing: PKB			✓			
		Unit Testing: PQL				✓		
		System testing for NextBip/NextBip* and AffectsBip/AffectsBip*		✓		✓		
		Extension documentation			✓	✓		
		Iteration 3 documentation	✓	✓	✓	✓	✓	✓

2.3.2 Iteration 3 Activities

This section describes the activities of our team for Iteration 3, which ran from Week 10 to Week 12.



Mini-Iteration	Activity	Thomas	Jonathan	Lim Li	Shawn	Dominique	Kenneth
1	Implement PKB support for Affects/Affects*			✓			
	Implement QueryParser support for Affects/Affects*				✓		
	Implement QueryValidator support for Affects/Affects*				✓		
	Implement QueryBuilder support for Affects/Affects*				✓		
	Optimization: QueryGrouper + QuerySorter					✓	✓
	Optimization: AC3	✓	✓				
	Other potential optimisation improvements	✓	✓		✓		
	Query Validator + Builder unit tests for Affects/Affects*				✓		
	Query Evaluator unit tests for optimization					✓	✓
	PKB unit tests for Affects/Affects*			✓			
	System tests for Affects/Affects*		✓		✓		
	System tests for optimization						
	System tests for advanced SPA requirements		✓		✓		
	Stress testing	✓	✓			✓	✓
	Plan extension for mini-iteration 2	✓	✓				
	Document extension					✓	✓
	Fix documentation based on TA feedback	✓	✓	✓	✓	✓	✓
2	Implement PKB support for NextBip/NextBip*			✓			
	Implement PKB support for AffectsBip/AffectsBip*			✓			
	Update Query Parser to support extension relations	✓					
	Update Query Validator to support extension relations				✓	✓	
	Update Query Builder to support extension relations				✓		✓
	PKB unit tests for NextBip/NextBip* and AffectsBip/AffectsBip*			✓			
	Query Validator unit tests for NextBip/NextBip* and AffectsBip/AffectsBip*				✓		
	Query Builder unit tests for NextBip/NextBip* and AffectsBip/AffectsBip*				✓		
	System tests for NextBip/NextBip*/AffectsBip/AffectsBip*		✓	✓			
	Document optimisation strategy					✓	✓
	Document extension			✓		✓	✓
	Finalise iteration 3 documentation	✓	✓	✓	✓	✓	✓

2.4 Rationale for Overall Plan

The rationale behind our plan is to ensure that we are able to implement the full SPA requirements and extension by the Iteration 3 deadline. On top of that, our plan ensures that we have sufficient time to test our solution extensively.

2.4.1 Coding

We decided as a team to have 1 person work on the SPA Front-End component, 2 people on the PKB component and 3 people on the Query Processor component. We decided on this division of labour based on how complex we felt each component would be.

At the start of each iteration, we held meetings to discuss the tasks that were to be completed by the end of the iteration. We then proceeded to break down the major iteration into a series of mini-iterations so that we can have a working solution throughout the iteration. This would increase motivation and allow us to detect bugs frequently and early.

This first mini-iteration of each major iteration is often used to fix code quality/perform refactoring of the entire code base. This allowed us to better implement new features in the subsequent mini-iterations. It also allowed us to identify underlying issues that were not identified in the previous iterations and resolve them before proceeding with the implementation of new features.

Each team was responsible for unit testing of their components and respective classes for each pull request. After which, we performed integration testing between the Front-End + PKB and PKB + Query Processor.

We aimed to complete all implementation and testing for each major iteration 1 week ahead of the deadline, leaving the last week as a buffer.

2.4.1.1 Development Tools

We used Continuous Integration (CI) to automatically build our program, run tests (unit tests, integration tests, system tests) and enforce code style standards on every pull request (PR). This was done on an Ubuntu and Windows environment as our team was cross-platform.

The use of CI within our version control platform was extremely convenient and allowed our work to flow very smoothly. Only PRs that passed all the CI tests would be merged, which ensured that each build was compatible with all members' operating systems.

2.4.2 Testing

Testing was conducted concurrently with coding in order to enforce correctness of the program incrementally. Throughout each mini-iteration, our test lead, Jonathan, would organise the system testing to ensure that we meet our goals set for each mini-iteration. Bugs found were immediately rectified before the start of the next mini-iteration. Our detailed test plan can be found in Section 7.

2.4.3 Documentation

Our documentation lead, Kenneth, would decide on the structure of the report to ensure that the documentation for each component is consistent. Each team was responsible for the documentation of their respective components following this structure.

We made sure to update the documentation during each mini-iteration instead of leaving it to the end of a major iteration.

3 SPA Design

3.1 Overview

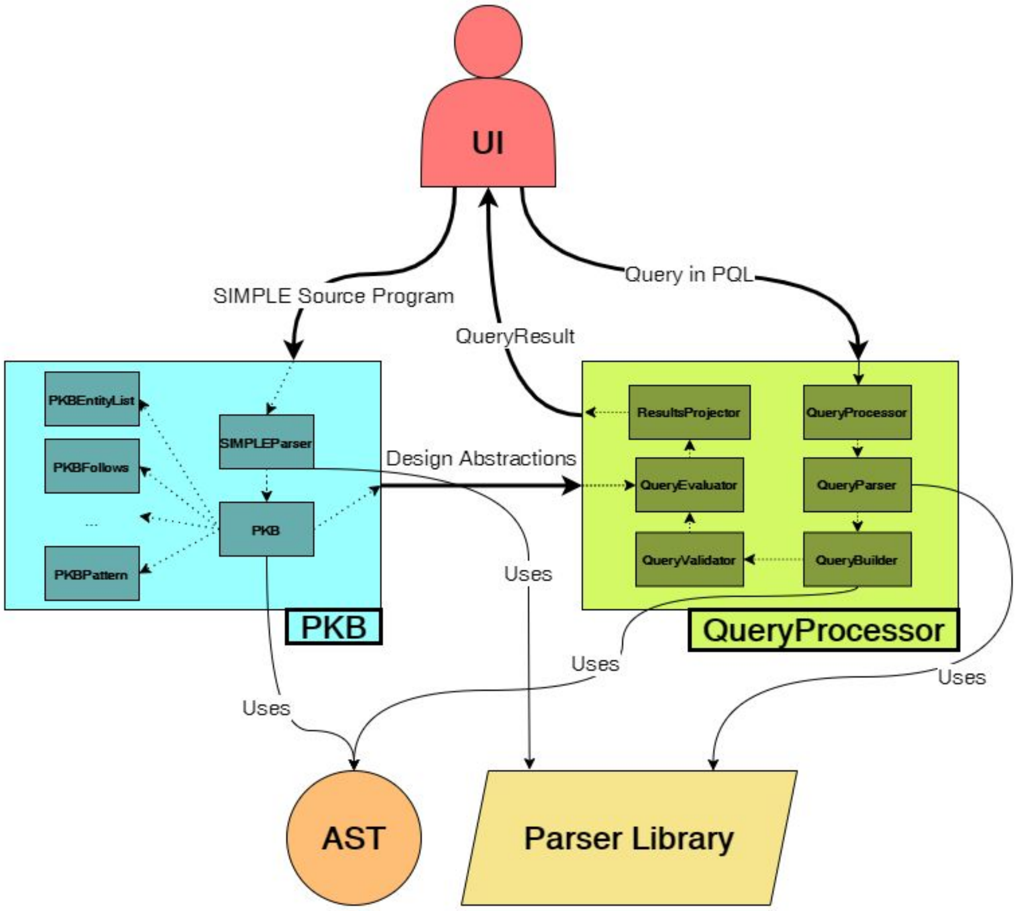


Figure 3.1.1: Architecture Diagram. Dotted lines are non-standard notation, roughly indicating the flow of data

The ***Architecture Diagram*** given by Figure 3.1.1 above explains the high-level design of our SPA. The SPA consists of two main components.

- **PKB**: Extracts design entities and design abstractions from the given SIMPLE source program and stores them in appropriate data structures. PKB exposes methods for the Query Processor to retrieve design abstractions to answer queries.
- **Query Processor:** Validates and evaluates user inputs queries written in PQL. To answer queries, Query Processor makes use of the program design abstractions stored in the PKB.

The two components are supported by two auxiliary components; a parser library and an AST data structure. These two auxiliary components require a much higher level of granularity to use, and thus simply exposes its structs/classes/functions as is.

- **Parser Library**: Aids the PKB and QueryProcessor in parsing SIMPLE source and PQL queries by exposing a library of parser combinators. The sub-components of the PKB and Query Processor that use the parser library are the SIMPLEParser and QueryParser respectively. While we call them sub-components, these components are not instantiated but expose a single static method to parse a SIMPLE program or PQL query string. These parse methods will return an AST as a result in both cases.
- **AST**: The AST data structure is abstracted into its own component because of its dual-use in both the PKB component and Query Processor, similar to the Parser Library. Traversal of the AST data structure, in either case, is done using the *Visitor Pattern.* Stores and supports traversal of a SIMPLE source program. The PKB uses it to store the result of parsing as well as to extract design abstractions. The QueryProcessor uses it to store the SIMPLE source within pattern clauses.

The following ***Sequence Diagram*** given by Figure 3.1.2 below shows high-level processes involving the PKB and Query Processor in a typical use-case of the SPA.

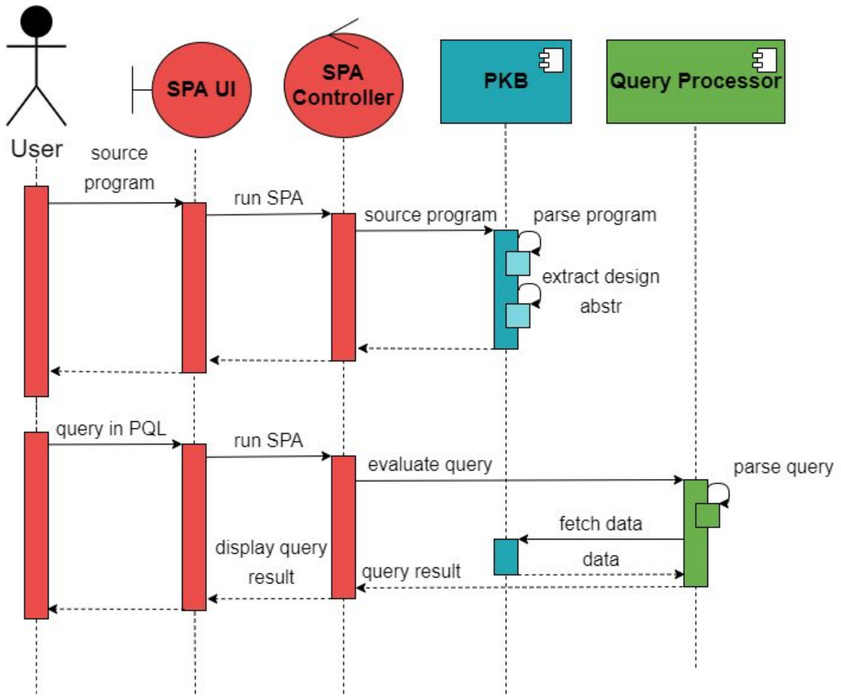


Figure 3.1.2: Component Interactions in SPA

3.1.1 Alternative SPA designs

We briefly considered an alternative more similar to the one suggested in lectures; where the PKB component is nothing more than a store for design entities and abstractions and most of the logic required to compute both the simple and transitive relationships are done in the SPA Front-End and Design Extractor.

The benefits of such a design is a dramatically simpler PKB component. All it needs to concern itself with is exposing API to insert relations and exposing API to retrieve them. This process is represented by Figure 3.1.1.1 below.

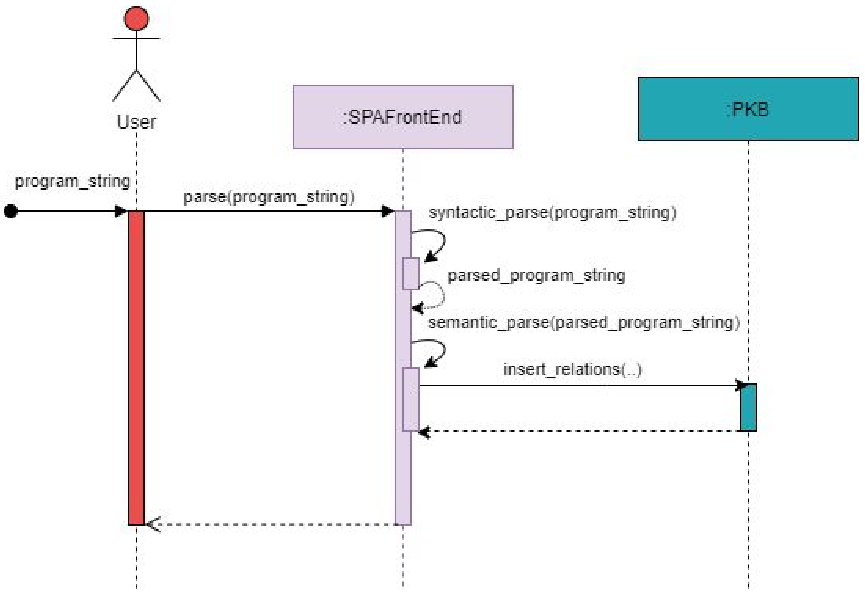


Figure 3.1.1.1: SPA Design as in the Lecture

However, this also forces the PKB to update its entries sequentially and reduces the opportunity for optimisations. Furthermore, this means that the responsibility of the SPA Front-End is more muddled; it needs to do parsing on the program string, and at the same time, extract design abstractions to set table values in the PKB. In other words, it needs to do both syntactic and semantic analysis on the program string, hence a lack of separation of concern.

To motivate more coherent components, we decided that the PKB should do both extraction and store design abstractions while the SPA Front-End is only concerned with the syntactic processing of the program string. To this end, an AST is more of a necessity than a design choice because the PKB component requires a flexible data structure that represents the SPA program to perform its algorithms and computations on. This process is represented by Figure 3.1.1.2 below.

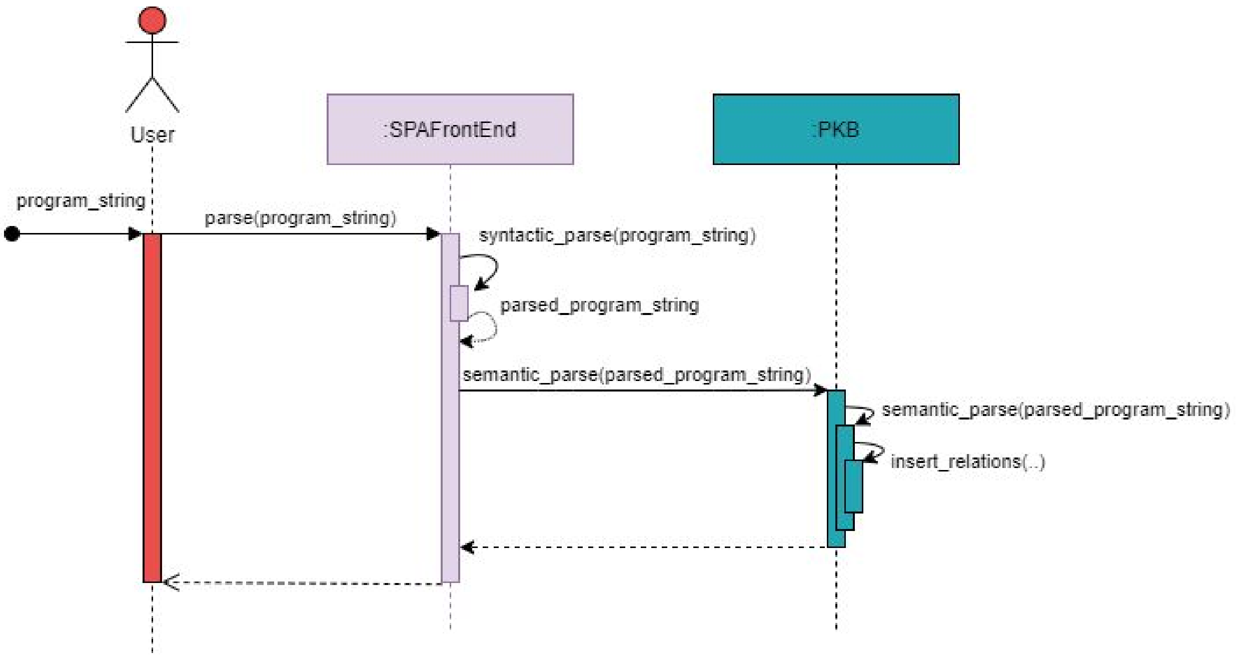

User
:SPAFrontEnd
:PKB
program_string
parse(program_string)
syntactic_parse(program_string)
parsed_program_string
semantic_parse(parsed_program_string)
semantic_parse(parsed_program_string)
insert_relations(..)

Figure 3.1.1.2: Alternative design. Note that "parsed_program_string" refers to the AST in our design

Notice how the SPA Front-End is just a parser; it delegates semantic parsing to the PKB. This is symmetric to how in the Query Processor, we have a component QueryParser to parse queries. Hence, we decided to treat the SPA Front-End as a sub-component of the PKB to mirror the way the QueryParser is a sub-component of the QueryProcessor. In the Figure 3.1.1.2 above, the SPAFrontEnd is synonymous with the SIMPLEParser component of our design in Figure 3.1.1.3 below, and belongs in the PKB component.

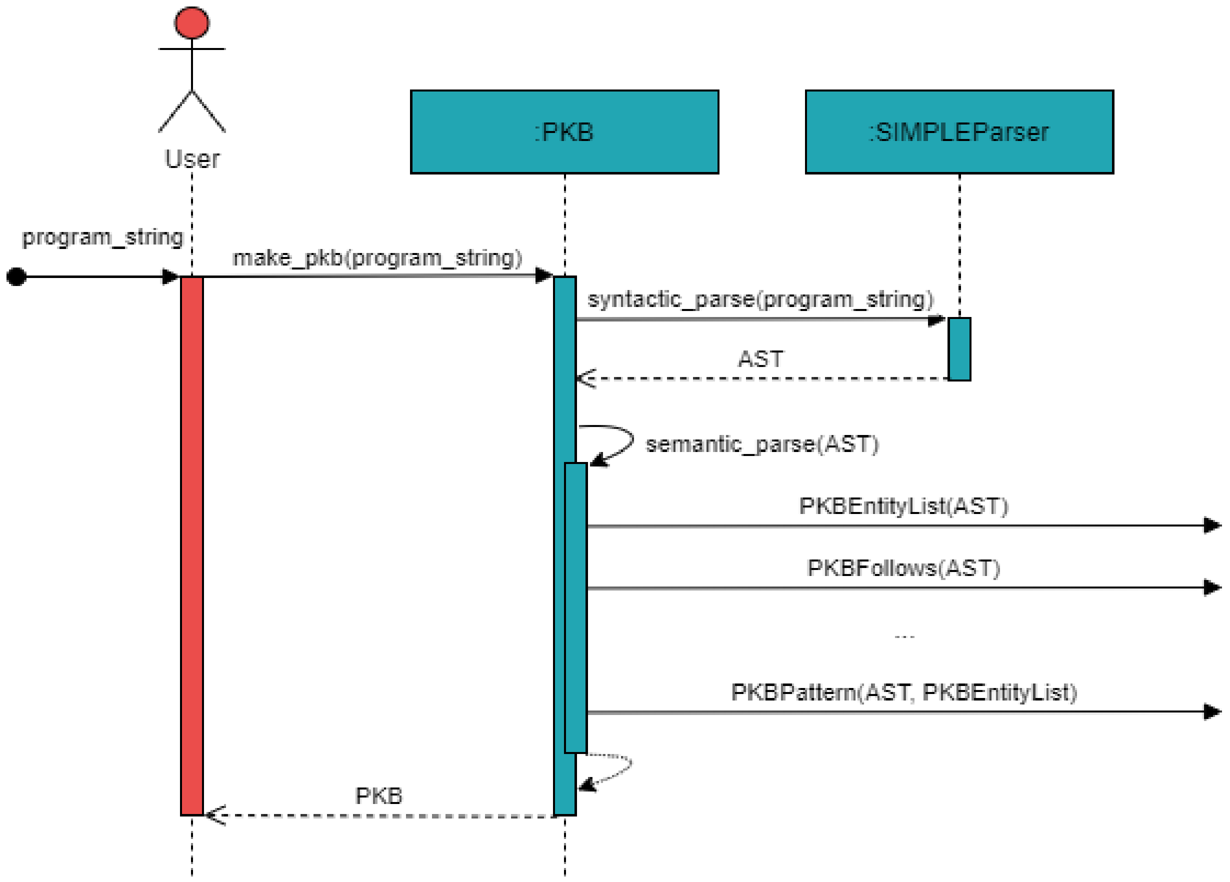


Figure 3.1.1.3. Chosen design. Refer to the [PKB section](#) below for more information about the population of PKB classes.*

This might appear as though we have decided to call the SPAFrontEnd a different name and merge the SPAFrontEnd and PKB, but that is not the case. As mentioned, the SIMPLEParser component in the PKB, while in name a sub-component of the PKB, is quite decoupled from the rest of the PKB sub-components. Similar to the QueryParser, it is not instantiated at all! It only uses the Parser library in the context of the SIMPLE language to return an AST.

3.2 Parser Library Component

The Parser library's responsibility is to simply expose an API to aid in parsing. This library is used in 2 places, the Front-End which parses SIMPLE programs, and the Query Processor which parses PQL queries.

In this section, rather than describing the design of the actual parser library itself, it is more important to discuss the design of what features the library implements at all, and how it helps in parsing SIMPLE source and PQL queries. Thus, we will be evaluating the design of the parser library through the eyes of the SIMPLE and PQL parsers.

It is also important to note that explanations and examples given in these sections will **not** be in the context of the SIMPLE language nor the PQL. A library is supposed to be as general as possible without sacrificing functionality. This Parser library is to be used in both the PKB and the Query Processor, so contextual explanations and examples will defeat the purpose of abstracting the parser functionalities out of either component. **For contextual explanations, refer to Section 3.4.1.1 and Section 3.5.1.2 for the PKB and Query Processor respectively.**

3.2.1 Parser Library Subcomponents

The parser library supports the construction of actual parsers. Using it, construction of parsers is done in a functional manner: Defining many short parsers that accept or reject a (series of) token(s), and chaining them together to form a larger parser. The library supports this by exposing several "combinators" to chain these short parsers together. More information about how it does so can be found in Section 3.2.2.4. Implicitly, the library also defines how parser functions (to be chained together) should be written.

It is then clear why abstracting this functionality out as a library is useful - we can now use the same parser library to implement parsers for the SIMPLE programming language and the PQL.

One more important note is that semantic validation (the checking of calls to non-existent procedures, cyclic and recursive calls) is not done here, but in the PKBCalls class. The reason for this is to respect the responsibilities of the components; the parser does syntactic parsing while PKB does semantic parsing.

3.2.1.1 ParserBase

This is the Abstract Base Class for all concrete parser implementations. It is a reification of a parsing function, and thus only needs to implement the following operator overload:

```
Optional<Pair<OutT, StateT>> operator()(StateT state);
```

3.2.1.2 Parser

Stores a pointer to a ParserBase. Because of the way C++ works, in order to correctly use the virtual table, we have to store a pointer. However, storing managed pointers to parsers directly makes it harder to call them, as you'd need to dereference the parser before calling it. This means we need to write `(*parser)(state)` instead of `parser(state)`.

The effect of the ParserBase and Parser classes is that now we can work with parsing functions through an abstract data type.

3.2.1.3 Parser combinators

The implementation of these combinators is mostly straightforward. They all involve either inheriting from ParserBase and implementing the parsing function appropriately, or use other parser combinators in its own definition.

We will say a given parser "succeeds" if it returns a value in the optional, and "fails" if it returns a null optional.

A more in-depth explanation of the types of the parser combinators with some of these type aliases:

```
ParseFunc<StateT, X> = (StateT state) → Optional<Pair<X, StateT>>
```

can be found in Appendix A - Parser Combinators.

3.2.2 Design Decision: Parser Library Component

The main design consideration was to reduce reliance on testing. Testing a parser can be a tedious and error-prone process. (Experience gained from all the parser bugs in the legacy bison Source parser used in Henz et. al.'s project, js-slang.)

This is because, in reality, we have to verify TWO separate issues when testing/verifying a parser. The first is that the parser itself operates correctly, and correctly parses the grammar it is designed to parse (implemented correctly). The second is that the grammar it is designed to parse is in fact the correct grammar (implementing the correct thing). While most bugs in the legacy Source parser were of the second form, because we are not allowed to use parser generators such as bison, bugs of the first form will be much likelier to occur as well.

3.2.2.1 Options Available

Functional recursive descent parser

The most straightforward design is to directly write a recursive descent parser, using parsing **functions** that call accept/peek functions. This is the way described in the Wikipedia article on recursive descent parsing.

However, this was rejected as there are many places where backtracking is required, which drastically increased the amount of boilerplate. Every function would need to copy the state, call its children parsers, add an if statement to check whether the child failed, and restore the state if it did. This fails the design consideration of having strong invariants. It is easy to forget to restore the state in a rarely used branch, which would result in a parsing bug.

Note that our chosen design only differs from the above in that we reify the functions, and add parser combinators, so it isn't that far off anyway.

String manipulation

Another design is to do intense string manipulation, splitting the string into parts by looking for occurrences of "procedure xxx {" to identify procedure boundaries, followed by looking for semicolons to identify statement boundaries. This was rejected as it does not fulfil the consideration of closely matching the grammar, which is described in BNF.

Pratt parser

The last design that was considered was a Pratt parser. However, Pratt parsers again do not match up with the description of the grammar, which is in BNF, and has already explicitly denoted the precedence rules with extra nonterminals. Using Pratt parsers to specifically solve left recursion (due to left associativity of the arithmetic operators) is probably pretty overkill. The last consideration was that the author has more experience with traditional parser combinator libraries than Pratt parsing.

3.2.2.2 Evaluation Criteria

The first is to establish **strong invariants** within the parser code so that we can be more easily certain that it is bug-free. The second is to design the internals of the parser to **closely match the grammar** it is designed to parse, so it can easily be compared to the SPA specifications. This way, the correctness of the parser can be reasoned easily by looking directly at the code.

A pragmatic concern that impacted the design was to accept designs that may take a long time to write upfront but minimizes future maintenance cost. This adds additional weight to the consideration that the parser should closely match the grammar, as changes to the grammar should be easy to make.

Another pragmatic concern is that the author has experience with parser combinator libraries, and so such solutions will come much more naturally than more imperative designs.

3.2.2.3 Evaluation of Options

	Design considerations	
	Maintains strong invariants	**Closely matches grammar**
Reified functional recursive descent parser	✓	✓
Functional recursive descent parser	✓	
String manipulation		
Pratt parser	✓	

3.2.2.4 Final Decision and Rationale

To enforce strong invariants on the parser, it is built in a **functional** manner. Each parsing unit starts its parse at a particular position in the string and returns a constructed object as well as the new position in the string if it succeeds, and an empty optional if it fails. In other words, the abstract type of a parser that parses an X is:

```
possible ParserState = (Index, Tokens, other state...)
Parser<StateT, X> : StateT → Optional<(X, StateT)>
```

For example, let's say we have such a parser function called `booyah_parser` that **accepts** if the current token is "5" by returning "booyah", and **fails** otherwise.

Then if

```
state.current_token() == "5"
```

we expect to see

```
booyah_parser(state) == std::make_optional("booyah")
```

and if

```
state.current_token() == "nobueno"
```

we expect to see

```
booyah_parser(state) == std::nullopt
```

This turns out to be sufficient to capture parsing: when we read a program, we expect to see certain tokens in certain orders. If we see them, we accept and return our understanding of the program. If we don't, we reject and say that we do not understand the given string.

By building a bunch of parsers that each understands different grammars, we can use this interface to compose parsers together and build even more parsers. For example, we can take a parser that's capable of understanding "_ + _" and combine it with another parser that's capable of understanding "_ - _", and now we can create a parser that's capable of understanding both.

This design does, however, limit the ways we can backtrack (once the Optional succeeds, there is no way to ask the same parsing unit to return another alternative). Luckily, this design is sufficient for SIMPLE, which can be parsed using a finite lookahead window.

We then **reify** the notion of this parsing function into a Parser class.

Parser combinators can then be built on top of this concept. For example, we can write a function that takes in a list of parsers and returns a parser that succeeds if any one of its parsers succeeds, returning the result of the first parser in the list that does.

```
make_alt : List<Parser<StateT, X>> → Parser<StateT, X>
```

To actually parse the tokens themselves, we can write a parser that succeeds if the current token is of a particular type, and fails otherwise. For example, for SIMPLE parsing we can:

```
make_token : SIMPLETokenType → Parser<SIMPLEParserState, SIMPLEToken>
```

We can now compose these functions to construct large parsers in a fashion that is very faithful to the grammar described in the SPA requirements.

Thus, we fulfil both design considerations as intended.

3.3 Abstract Syntax Tree

As mentioned, the PKB will now do a semantic analysis of the SIMPLE program. This means that we require an easy-to-use data structure to retrieve information about the SIMPLE program. This is where the AST comes in; the AST component's responsibility is to store SIMPLE programs, and importantly, expose an API for traversal of the AST. To illustrate, these are some of the traversal requirements of some design abstractions:

- Extracting all statements of particular types (if/while/etc.)
    - Requirement: Ability to extract nodes of specific types.
- Modifies (Assignment), Uses (Assignment), pattern

    - Requirement: Ability to extract assignment nodes, and retrieve their contents, and traverse the expression contained.
- Follows(*)
    - Requirement: Ability to extract statement list nodes and retrieve their contents.
- Parent(*)
    - Requirement: Ability to extract if/while nodes and retrieve their contents.

The API exposed allows the PKB to define “type-aware functions” (“type” referring to type of AST node) to apply on each node of the AST. The AST implements the machinery required to make sure that the provided functions are applied to each node of the AST in order. Hence the PKB (or any other component that wants to traverse the AST) has a much simpler job, just define the functions to be applied and pass it to the AST. A detailed explanation follows.

3.3.1 Visitor pattern

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. As a result of this separation, we would be able to add new operations to the AST data structure without modifying the AST. It is one way to follow the open/closed principle.

In the visitor pattern, traversal through the AST is done through an intermediary “Visitor” derived class. The AST node accepts the “Visitor” class and applies itself to the “Visitor” class’s “visit” function. After which, depending on the boolean output of the “visit” function, it passes the same “Visitor” class to its child nodes, and the process recurses downwards.

Each AST has an accept(visitor) method, and it would call visitor.visit(). If the visit() function returns false, then we would recurse down the AST tree. Else, the visitor would stop.

This is the pseudocode of the accept() method for the AST:

```
accept(visitor):
      shoud_stop = visitor.visit(this)
      if (should_stop):
            return
      for (child of this):
            child.accept(visitor)
```

And this is the pseudocode for a visitor that extracts all Read statements:

```
ReadVisitor {
      list = []

      visit(ast):
            if (ast is a Read statement):
                  list.add(ast)
            if (ast is an Expression or a Print/Read/Assign statement):
                  return true   // stop recursion
            else:
                  return false  // continue recursion
}
```

The `ReadVisitor` overwrites the overload-ed `visit()` function, The root AST node `p` accepts the visitor and recursively passes down the `ReadVisitor` to its child AST nodes if and only if the node is a `Stmt` type AST node (where the `visit()` function will return false). At each `Stmt` node visited, the node applies itself to the `ReadVisitor`'s `visit()` function. Only when the node is a `ReadStmt` will the overwritten `visit()` record the statement.

The strength of the Visitor pattern is clear - the algorithm of the type-aware traversal (logging `ReadStmt`'s) is separate from the data structure (the AST). This allows for ease of implementation of different traversal algorithms; simply inherit from the appropriate Visitor class and overwrite the appropriate `visit()` functions where needed.

For example, in this simple program and its corresponding AST given by Figure 3.3.1:

```
procedure main {
1.     while (1 == 1) {
2.           a = 2;
3.           read a;
       }
}
```

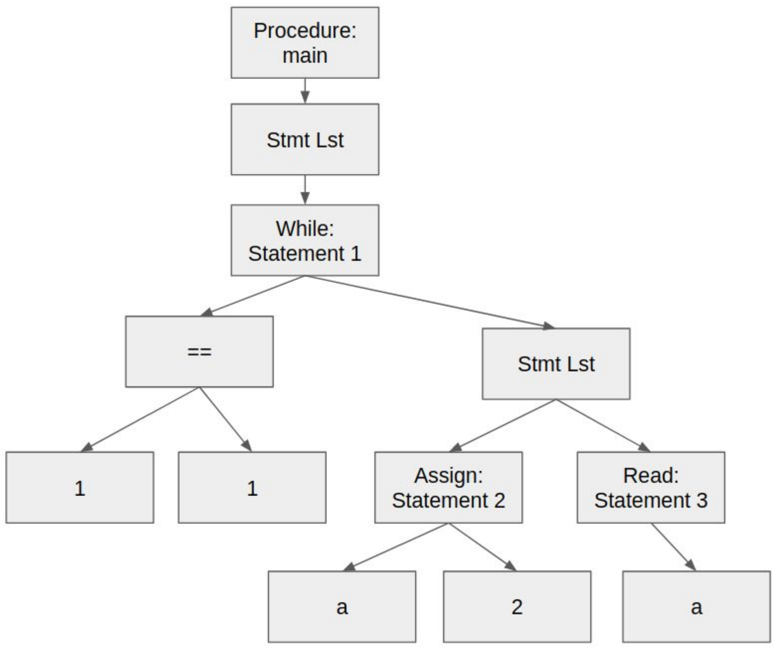


Figure 3.3.1: AST of Sample Source Program

Suppose we would like to extract all Read statements. We could define the `visit()` function as above, where if it is visiting a Read statement, it would add it to a cumulative list. Else, it will either stop or recurse down the AST tree. Hence, the visitor would visit all the nodes of the AST in a DFS order, and Statement 3 would be added to the cumulative list. Hence, the nodes that would be visited are given by Figure 3.3.2 below.

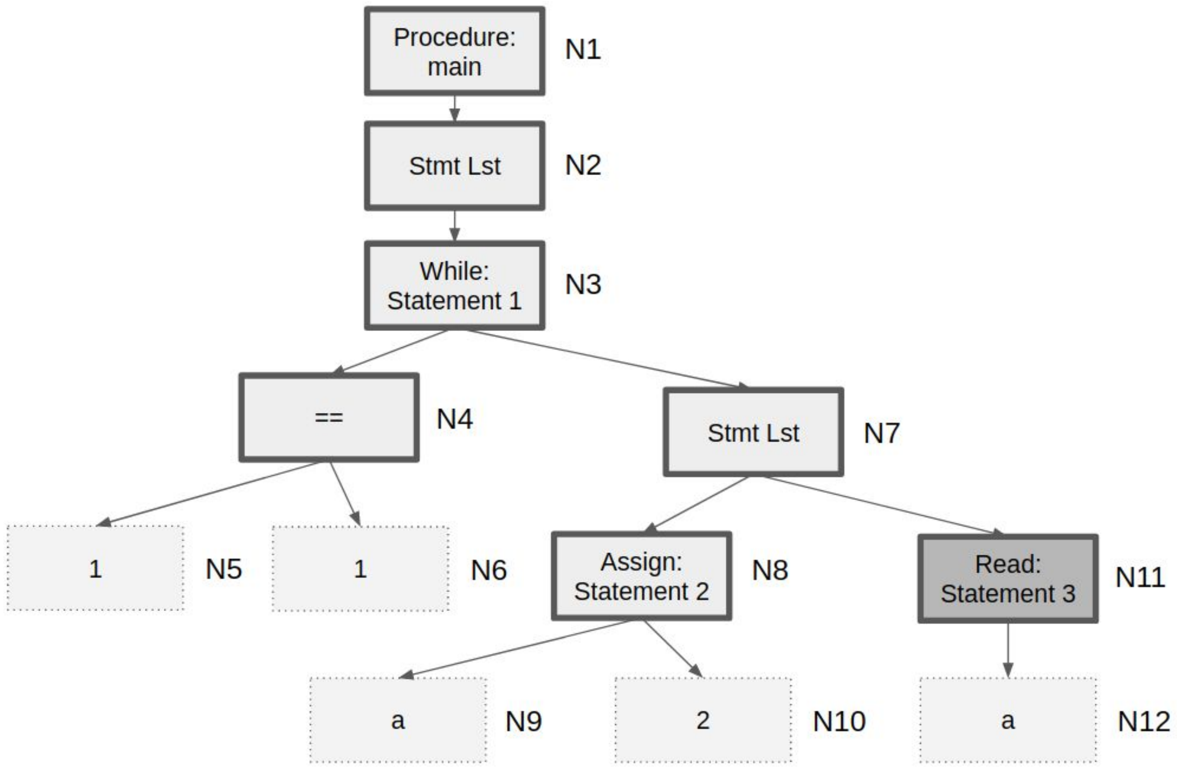


Figure 3.3.2: Visited Nodes to Visit Read Statement

The nodes with solid borders are nodes visited by the visitor, and the nodes with dotted borders are not visited. The Read statement is darker in colour, denoting that it is added to the global list. The nodes are labelled N1 to N12 for ease of explanation.

This would be the simulation of the ReadVisitor:

- N1.accept(visitor)
- visitor.visit(N1)
    - N2.accept(visitor)
    - visitor.visit(N2)
        - N4.accept(visitor)
        - visitor.visit(N4)
            - Stop here because visit() returns true
        - N7.accept(visitor)
        - visitor.visit(N7)
            - N8.accept(visitor)
            - visitor.visit(N8)
                - Stop here because visit() returns true

- N11.accept(visitor)
- visitor.visit(N11)
    - Add N11 to list, and stop here

3.3.2 Design Decision: AST

We were aware of existing solutions to the problem by Javascript's acorn-walk, and Python's ast module. As both Javascript and Python are dynamically typed languages, there is a slight difference, but both employed the Visitor pattern to carry out the traversal, and it has been found to be a very flexible and accurate solution to the problem of AST traversal in the real world. This adds bonus weight to solutions that use the Visitor pattern to support traversal if the additional technical complexity does not outweigh the benefit.

Ultimately, because all the designs we considered easily supported the Visitor pattern, all our solutions used it.

3.3.2.1 Options Available

Inheritance-based Class Hierarchy + Visitor pattern

This design creates a class inheritance hierarchy that closely matches the ASG described in the SPA requirements. Abstract Base Classes are used for parts of the ASG where the disjoint union is required, with the child classes inheriting from these Abstract Base Classes.

For example, for the following disjoint union in the ASG, we define an Abstract Base Class:

stmt: read | print | call | while | if | assign

```
struct Stmt; /* Forward declarations */
struct ReadStmt;
...

/**
 * Abstract Base Class for stmt
 */
struct Stmt : AST { /* ... */ };

struct ReadStmt : Stmt { /* ... fields */ };

...
```

Composition-based class hierarchy enabled by `std::variant` + Visitor pattern

Instead of using an inheritance-based class hierarchy to represent the nonterminals of the ASG, we could also use a composition-based class hierarchy, where classes corresponding to nonterminals *contain* its respective child classes. `std::variant` is used to take the disjoint union of child classes.

Using the same example:

```
struct Stmt; /* Forward declare so it can be contained in WhileStmt */

struct ReadStmt { /* ... fields */ };
...

struct Stmt {
  std::variant<ReadStmt, PrintStmt, ...> stmt;
};
```

The downside of this approach is that it does not actually work. `std::variant` requires all its template arguments to be complete before it can be used. One way to get around this issue is to take the disjoint union of a managed pointer, which is a complete type even if its template arguments aren't.

```
struct Stmt; /* Forward declare so it can be contained in WhileStmt */

struct ReadStmt { /* ... fields */ };
...

struct Stmt {
  std::variant<std::unique_ptr<ReadStmt>, ...> stmt;
};
```

While this approach should have worked, it didn't. C++ is a mystery.

Handwritten composition-based class hierarchy + Visitor pattern

To resolve the above problems with properly using the C++ standard library, we considered using a hand-written disjoint union instead of using C++'s templated classes. The need to handwrite disjoint unions increases the implementation difficulty. The visitor pattern can work in essentially the same way, although the need for double dispatch is now reduced.

A double-edged sword of using a composition-based class hierarchy in this fashion is that the types are now stronger, as there is no base class that can store any type of node in the AST. This makes type aware traversal less error-prone, but it does make traversal itself annoying to write. This further complicates the implementation.

3.3.2.2 Evaluation Criteria

One design consideration is to **closely match the Abstract Syntax Grammar (ASG)** described in the SPA requirements. This is because the rest of the SPA requirements frequently reference the ASG, with concepts such as “statement”, “procedure”, and “subexpression” being used to describe the semantics of design abstractions such as “Modifies”, “pattern”, and so on.

Another design consideration is the ease of traversal of the AST to extract the necessary information to construct design abstractions. This is because we need to traverse the AST several times in the construction of the PKB, and different design abstractions require different traversal algorithms (as mentioned above).

It is clear that the following is sufficient to satisfy all of the above requirements:

- Sufficient to have: Ability to extract nodes of any type, and retrieve their contents depending on their type.

Or in other words, **type-aware traversal**.

We also prioritised that the AST should **not be too difficult to implement**, with all else being equal.

3.3.2.3 Evaluation of Options

	Design considerations			
	Closely matches grammar	**Type-aware traversal**	**Works**	**Implementation Difficulty (1-10)**
Inheritance-based class hierarchy + Visitor pattern	✓	✓	✓	5
Composition-based class hierarchy enabled by `std::variant` + Visitor pattern	✓	✓		7
Handwritten composition-based class hierarchy + Visitor pattern	✓	✓	✓	8

3.3.2.4 Final Decision and Rationale

Inheritance-based Class Hierarchy

In order to support traversal and extraction of nodes of any type, and retrieve contents depending on the type of the node, we implemented a double-dispatch Visitor pattern, which is perfectly suited for such inheritance-based class hierarchies and makes use of C++’s virtual function table to carry out the dispatch.

3.4 PKB Component

The PKB component's responsibility is to parse and extract design entities and design abstractions from the SIMPLE source program, and subsequently answer queries for these design entities and abstractions made by the Query Processor component via the PKB API.

As a quick overview, the main PKB class mainly acts as a router; delegating the jobs of extracting and retrieving specific entities/relations to other PKB subclasses, e.g. PKBFollows, PKBEntityList. Figure 3.4.1 below is a class diagram of all sub-components of the PKB:

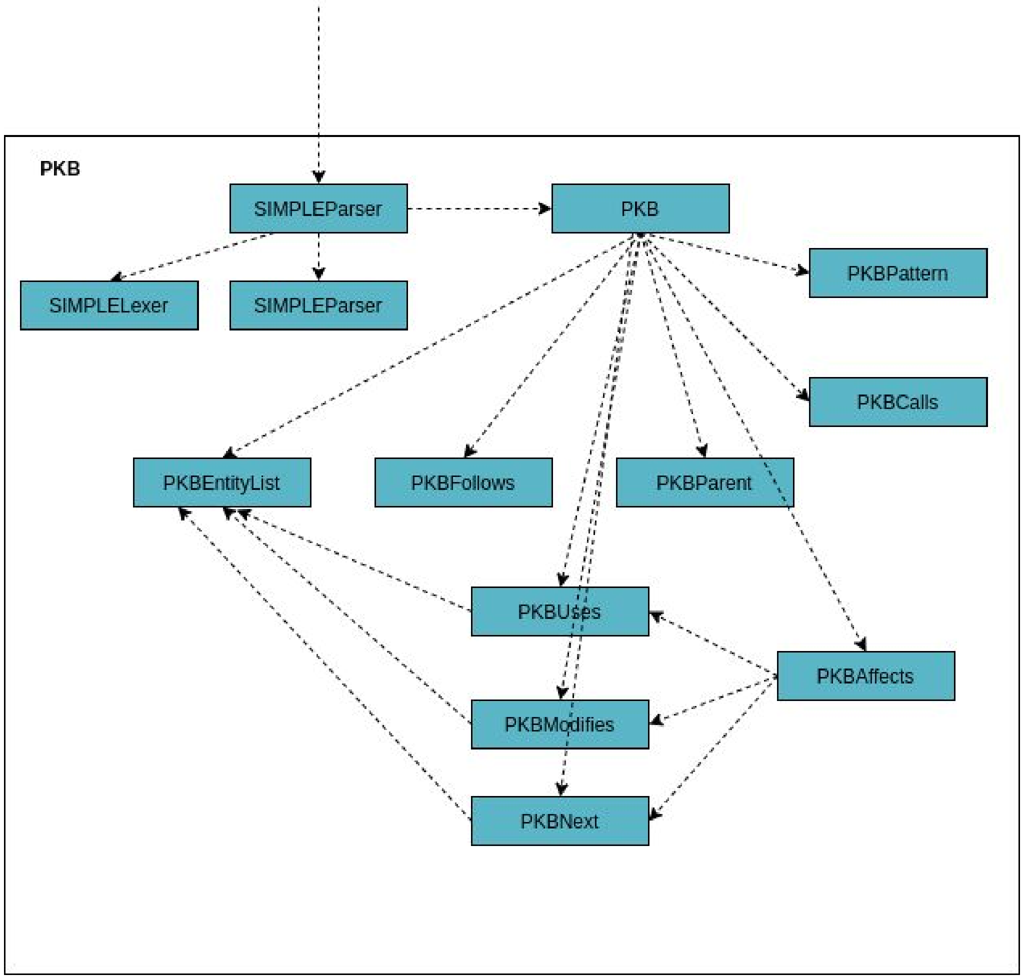


Figure 3.4.1: PKB Sub-Components

To this end, the main PKB class does not directly contain information of the design entities and abstractions, and only contain the subclasses and the AST. The following diagrams serve as an overview for both extracting and retrieving design entities and abstractions but should be taken in the context of the rest of this section.

We also make a distinction between PKB **component** and PKB **class**. This distinction is important when taking the SIMPLEParser component into consideration since both the PKB class and SIMPLEParser are sub-components of the PKB component, but the SIMPLEParser is decoupled from the PKB class.

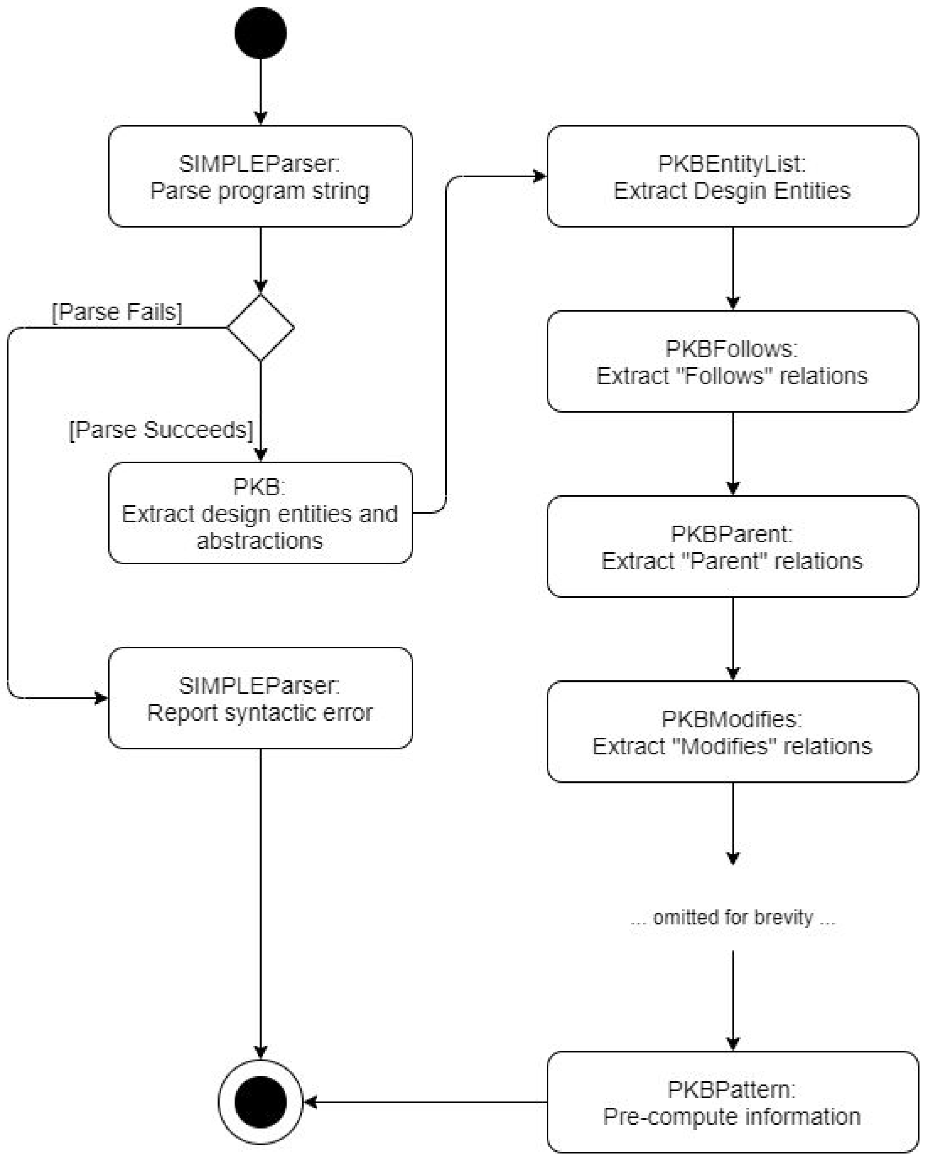


Figure 3.4.2: Activity Diagram for Extracting Design Entities and Abstractions

The above activity diagram shown by Figure 3.4.2 describes constructions of the PKB sub-classes. Note that **there are no API calls for extracting design entities and design abstractions**. The PKB subclasses perform their own traversal on the input AST when they are constructed, and fill its own data-structures containing all the relations.

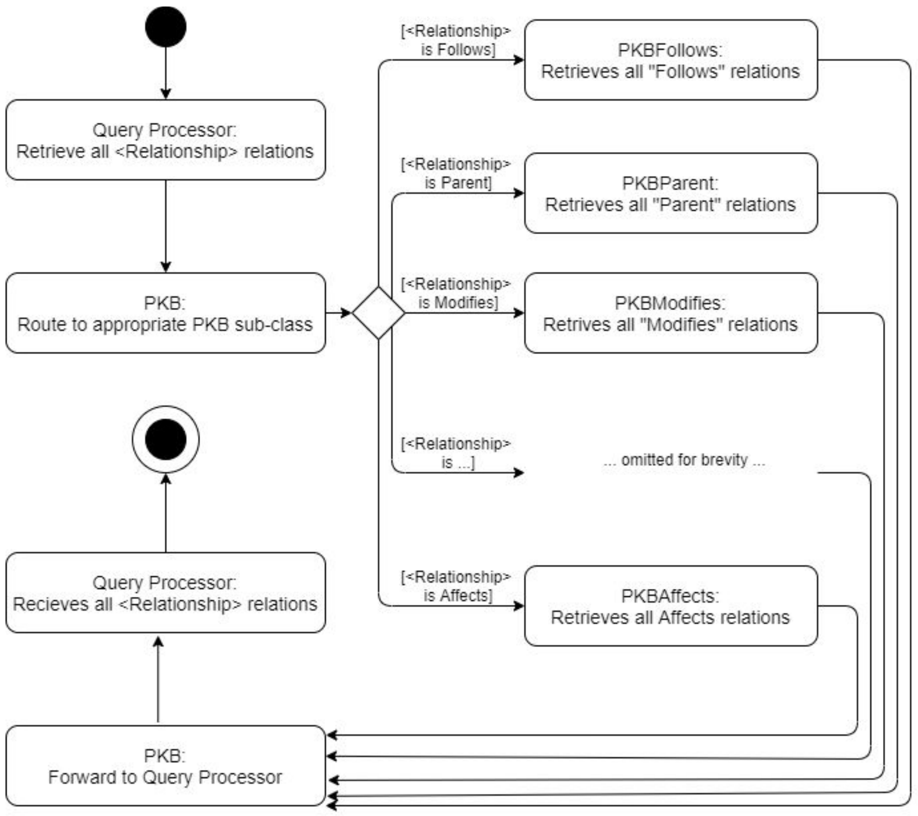

Query Processor:
Retrieve all <Relationship> relations
PKB:
Route to appropriate PKB sub-class
[<Relationship> is Follows]
PKBFollows:
Retrieves all "Follows" relations
[<Relationship> is Parent]
PKBParent:
Retrieves all "Parent" relations
[<Relationship> is Modifies]
PKBModifies:
Retrives all "Modifies" relations
[<Relationship> is ...]
... omitted for brevity ...
[<Relationship> is Affects]
PKBAffects:
Retrieves all Affects relations
Query Processor:
Recieves all <Relationship> relations
PKB:
Forward to Query Processor

Figure 3.4.3: Activity Diagram for Retrieving All Relations for a Particular Relationship

The above activity diagram shown by Figure 3.4.3 describes how the PKB class acts as a router to specific PKB sub-classes to retrieve or test relations. This means that the PKB class is able to expose only general API to the PQL. Refer to section 9.3 PKB API for a list of general API exposed.

In the following sections, we describe each of the PKB sub-components in detail. Also, note that mentions of traversal of the AST refer to using the visitor pattern described in Section 3.3.1.

3.4.1 PKB Sub-components

The PKB Component is made up of the following classes/sub-components:

- **SIMPLE Program Parser**:
  - **SIMPLELexer**: identifies contiguous sequences of letters/digits (names), contiguous sequences of digits (integers), symbols, etc. while discarding all whitespace
  - **SIMPLEParser**: performs syntactic analysis on the given PQL and returns a syntactically validated program AST
- **PKB Data Structures**:
  - **PKBEntityList:** extracts and stores the different design entities in their own list

- **PKBFollows:** extracts all Follows(*) relationships between statements
- **PKBParent:** extracts all Parent(*) relationships between statements
- **PKBUses:** extracts all Uses relationships between entities
- **PKBModifies:** extracts all Modifies relationships between entities
- **PKBPattern:** answer queries regarding patterns
- **PKBCalls:** extracts all Calls(*) relationships between procedures
- **PKBNext**: extracts all Next(*) relationships between statements
- **PKBAffects**: answers queries regarding Affects

The SIMPLELexer and SIMPLEParser are used to parse and analyze the SIMPLE program. After parsing the SIMPLE program, an AST will be created and passed to the PKB for each subcomponent to do further processing for each specific relationship.

Each class for the PKB Data Structures stores relevant information for each relationship in the query and provides an API to query relationship status between object entities. For example, in the PKBFollows class, an API is provided to query if two statements Follows or Follows* each other. Other data structure classes provide similar APIs for different relationships, and each class will be further elaborated below.

3.4.1.1 SIMPLE Program Parser

In order to simplify reasoning, we split lexical analysis and syntactic analysis into two components, the lexer and the parser. The lexer converts the program string (a list of characters) into a list of tokens, and the parser converts the list of tokens into a program AST.

SIMPLELexer

The lexer identifies contiguous sequences of letters/digits (names), contiguous sequences of digits (integers), symbols, etc. while discarding all whitespace. The full list of token types are rather comprehensive and omitted in this report. Instead, we describe how the program is lexed.

The lexer performs one character lookahead to tokenize the program string, scanning through all whitespace between valid tokens. Take the example below for how a typical assignment statement is tokenized:

```
x1s4variable = (14 / (y));
```

The lexer encounters the “x” alphabet character, and scans all alpha-numeric characters until it sees the space, which isn’t alpha-numeric, and stops. This identifies “`x1s4variable`” as a “`NAME`” token.

The next valid tokens are “`=`” and “`(`”, tokenized to “`ASSIGN`” and “`LPAREN`”. Note that the decision to tokenize “`=`” as the assignment token can only be made after peeking the next character and verifying it is not another "`=`" character (in which case the lexer will tokenize them both as the "`EQ`" token).

It then encounters "`1`", and proceeds to scan all digits to form an "`INTEGER`" token from the string "`14`". If the lexer encounters a non-whitespace and non-digit character while attempting the lex an integer, for instance, "`1a4`", it reports an error and aborts.

In this way, the SIMPLELexer tokenises the whole program string into a list of tokens with which the SIMPLEParser can parse on. The list of tokens for the above one-line program is shown in Figure 3.4.1.1.1 below:

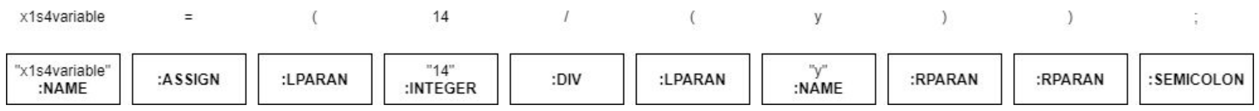


Figure 3.4.1.1.1: List of Tokens After Lexing

Furthermore, because only valid tokens are accepted, this doubles as a part of syntactic **validation** – any invalid tokens will result in an error.

SIMPLEParser

This sub-component performs syntactic analysis on the given PQL and returns a syntactically validated program AST.

The parser is a standard parser written using parser combinators, in a fashion that closely mirrors the grammar as described in the SPA requirements.

This involves looking at the grammar and mechanically translating it into calls to the parser combinator. To illustrate, Figure 3.4.1.1.2 below shows the key steps in **creating a parser for procedures**.

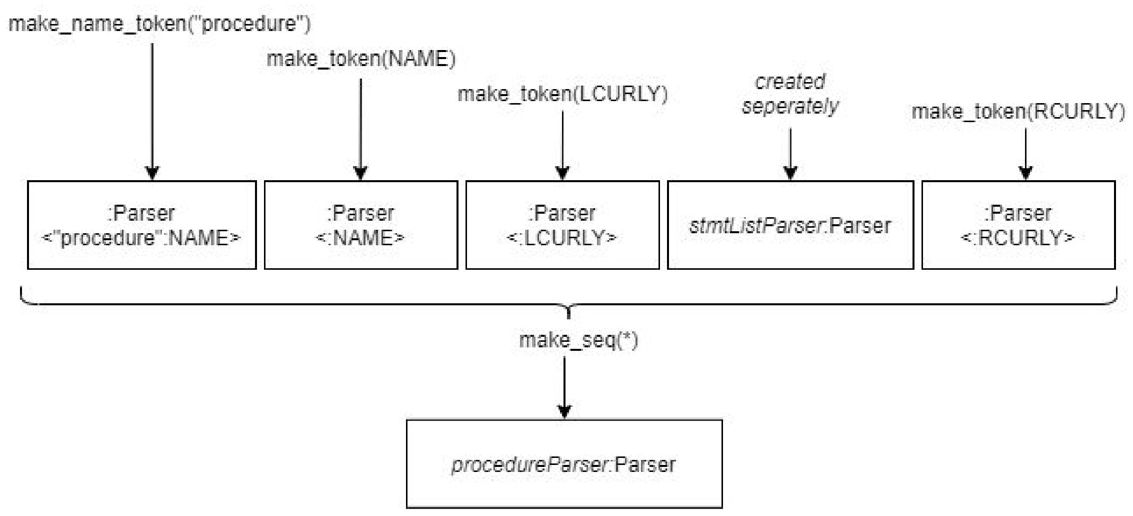


Figure 3.4.1.1.2: Steps in Creating Parser for Procedures

First of all, we need to define a parser that can check for particular tokens provided by the lexer and either accept them or fail. Once this is done, we can use the parser combinators to compose larger parsers.

We do this using the `make_token` function, which is implemented using the `make_fun` utility, which lets us convert a lambda function into a Parser object.

```
// type indicates the type of tokens we want to accept
Parser_<SIMPLEToken> make_token(Type type, ExpectedToken expectedToken) {
  return make_fun<SIMPLEParserState, SIMPLEToken>(
      [...](auto state) -> ... {
        if (state.cur().type == type) {
          // If the current position of the parse
          // is at a token that we want to accept
          return state.step(); // accept it by moving the state forward
        } else {
          state.report_error(state.curPos, expectedToken);
          return std::nullopt; // fail by not moving the state
        }
      });
}

// Similarly, we define a function make_name_token that accepts only
// if both the type of the token is NAME and its contents match
// a given string.
```

Once we can parse single tokens, it is relatively easy to compose them together to form larger parsers.

For example, we have the rule

procedure: ‘procedure’ proc_name ‘{‘ stmtLst ‘}’

To create a parser from this rule we simply mechanically transform it into code (note that the actual implementation is slightly more compact):

```
Parser<..., Tuple<Token, Token, Token, StmtLst, Token>> procedure1 =
    make_seq( // Combine a sequence of parsers
      make_name_token("procedure"),   // Parser for 'procedure'
      make_token(NAME),               // Parser for any name
      make_token(LCURLY),             // Parser for '{'
      make_non_owning(stmtLstParser), // Parser for stmtLst
      make_token(RCURLY)              // Parser for '}'
    );
```

The above understand the grammar, but the output type is wrong, returning a tuple of its parts instead of a Procedure AST node. Using make_lift, we can correct the output type to a Procedure.

```
Parser<..., Procedure> procedure =
    make_lift(std::move(procedure1),
              [](auto procedure_parts) -> Procedure {
                return Procedure{get<1>(procedure_parts),  // name
                                 get<3>(procedure_parts)}; // stmtLst
              });
```

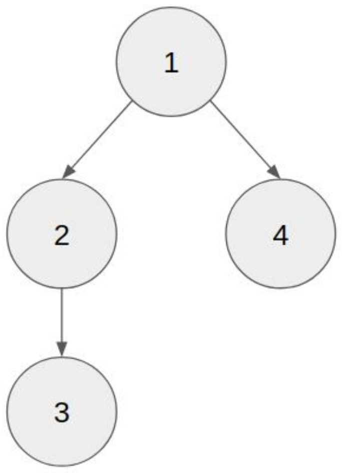
We simply do this for every single rule in the entire grammar, using combinators such as `make_seq`, `make_alt`, etc. to parse the grammar, and using `make_lift` to convert them to AST nodes.

Furthermore, because every parser only accepts if it sees something that's syntactically valid, the Parser simultaneously performs **validation** with no additional effort.

3.4.1.2 ConnectivityChecker

This is an auxiliary data structure class that can be used to precompute transitive relations of a graph and provides an API to check if two nodes are connected either directly or indirectly. Since checking for transitive relations is common in multiple PKB operations, an auxiliary class is created to reduce code duplication. This class is also used to store relations in PKBParent, PKBUses, and PKBModifies.

The number of nodes, as well as the list of edges, will be provided to the ConnectivityChecker. Then, the ConnectivityChecker would create an adjacency matrix and populate it with any direct edges. For example, figure 3.4.1.2.1 shows how a graph will correspond to its adjacency matrix:

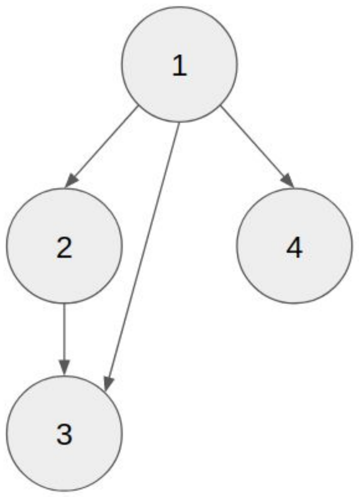


	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	0	0	0

Figure 3.4.1.2.1: Graph and Its Corresponding Adjacency Matrix

The matrix contains a 1 iff there is a direct edge connecting the nodes

Subsequently, all transitive relationships are computed using the Floyd Warshall Algorithm and stored in another adjacency matrix. The transitive relations for the above example is given by Figure 3.4.1.2.2 below.



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	0	0	0	0

Figure 3.4.1.2.2: Transitive Relations

The matrix contains a 1 iff there is a transitive edge connecting the nodes

Hence, the data structure would be able to support queries asking for direct or indirect relations and can answer them in O(1) time complexity.

The adjacency matrix is stored `std::vector<std::vector<bool>>`. Mapping between design entities and indexes are stored elsewhere inside the PKB subclasses.

3.4.1.3 PKBEntityList

This class extracts and stores the different design entities in their own list. It performs a walk through the entire AST once, adding the current node to the appropriate list at each step. Its internal API offers to fetch any of these lists and fetch particular procedure or statement AST node from the procedure name or statement number.

For instance, in this program:

```
procedure main {
1.    while (1 == 1) {
2.          a = a;
3.          print a;
      }
4.    b = 5;
5.    a = a + 1;
6.    print b;
}
```

Each statement's AST tree would be stored (as a pointer) in a list for easy retrieval.

Index	AST stored
0	AST of statement 1
1	AST of statement 2
2	AST of statement 3
3	AST of statement 4
4	AST of statement 5
5	AST of statement 6

For each statement type, the list of statements of that type is also recorded:

While statements
Statement 1

Print statements
Statement 3
Statement 6

Assign statements
Statement 2
Statement 4
Statement 5

The other lists (list of read statements, list of if statements, list of procedures, etc) are similarly stored and have been omitted for brevity. The full set of lists stored are as follows:

- List of statement AST's; indexed by statement number
- List of While statement numbers
- List of If statement numbers
- List of Print statement numbers
- List of Call statement numbers
- List of Assignment statement numbers
- List of Variable names
- List of Procedure names
- List of Constant values

Hence, to provide the list of print statements, the contents of the list of print statements could be simply provided without the need to iterate through the entire program again. To obtain the AST of statement 5, the list of statement ASTs could also be checked at index 4 without the need to iterate through the entire program.

All these lists are stored using `std::vector<*>`, where `*` refers to the relevant type of entity stored.

3.4.1.4 PKBFollows

This class extracts all Follows(*) relationships between statements. Since Follows(*) relationships only exist between statements in the same statement list, it extracts and maintains lists of pairs (statement number and statement list index) for each statement list in the AST. The class also maintains a map from statement number to the list of pairs in which it uniquely belongs. Checking whether two statements satisfy a Follows relation (Follows(s1, s2)) is then equivalent to consulting the map and verifying that:

1) Both statements belong in the same list of pairs
2) The statement list index of s1 is one smaller than the statement list index of s2

Checking Follows* relations are similar, except at the second check above we only require the statement list index of s1 is strictly smaller than that of s2.

For instance, in this program:

```
procedure main {
1.    while (1 == 1) {
2.           a = a;
3.           print a;
      }
4.    a = 5;
5.    a = a + 1;
}
```

The visitor would visit every statement list in the program. For each statement list, all statements in that list will be recorded and mapped to an ID unique to each statement list. In this case, the main procedure's statement list will contain statements 1, 4, 5, and the while loop's statement list will contain statements 2, 3. Hence, Figure 3.4.1.4.1 below shows the map data structure stored.

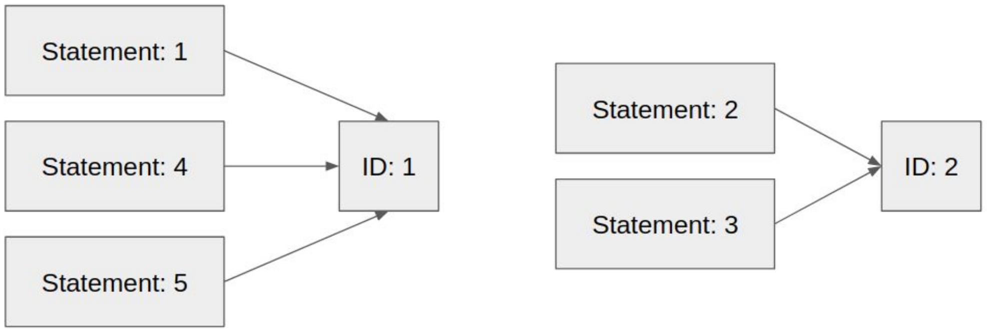


Figure 3.4.1.4.1: Data Structure of Statements

For example, if we want to query Follows(1,2), the statement list ID of statements 1 and 2 will be checked to be ID 1 and ID 2 respectively. Since they are not equal, hence, Follows(1,2) is false. If we want to query Follows*(1,5), the statement list ID of statements 1 and 5 will be checked and both have ID 1. Since they are equal, and 1 is smaller than 5, hence, Follows*(1,5) is true.

These maps are stored using

```
std::map<StmtNo, std::pair<std::shared_ptr<std::vector<StmtNo>>, short>>
```

to map each statement number to the lecture that contains it, as well as its index in the vector.

3.4.1.5 PKBParent

This class extracts all Parent(*) relationships between statements. This is done by filling two adjacency matrices for two directed graphs, one for direct Parent relationships and another for transitive Parent* relationships. These graphs are stored in an auxiliary class `ConnectivityChecker`. It first extracts an edge list of all direct Parent relationships from the AST, adding an edge for each statement in the statement list of an if or while AST node. `ConnectivityChecker` accepts this edge list as input to its constructor to generate the adjacency matrix for direct Parent relationships, and then subsequently uses the Floyd Warshall algorithm to compute the adjacency matrix for indirect Parent relationships.

Checking whether two statements fulfil a Parent(*) relationship is then a matter of consulting the appropriate adjacency matrix to verify there exists an edge.

For instance, figure 3.4.1.5.1 below shows a sample program and its corresponding graph.

```
procedure main {
1.      if (1 == 1) then {
2.              while (1 == 1) {
3.                      a = 1;
                }
4.              b = 1;
        } else {
5.              c = 1;
        }
}
```

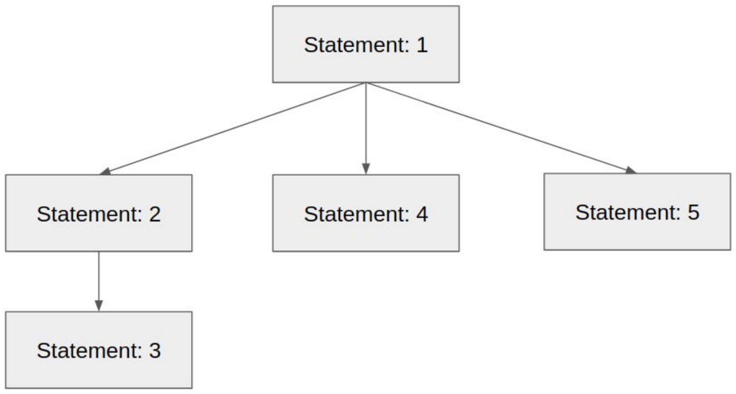


Figure 3.4.1.5.1: Graph for Parent(*) Relationship

The statement 1 would be a parent to statements 2, 4, 5, and statement 2 would be a parent to statement 3.

This graph would then be passed into `ConnectivityChecker`, so that `PKBParent` does not need to handle the actual storage of the graph, or the logic of checking transitive relation, as this has been abstracted out and will be handled by the `ConnectivityChecker`. The `PKBParent` simply has to query the data structure.

There are no data-structures stored inside the PKBParent besides `ConnectivityChecker`.

3.4.1.6 PKBUses

This class extracts all Uses relationships between entities. This is done by filling an adjacency matrix for a directed graph. The nodes of the graph are all statements, variables and procedures. An edge from a statement/procedure to a variable node implies that the statement/procedure uses that variable. Similar to PKBParent, this adjacency matrix is generated and stored in the auxiliary `ConnectivityChecker` class.

The class first walks through the AST, drawing edges from statements to variables where the statement directly uses the variable, and also edges between statements and procedures to represent a possible path for the Uses relationship. For instance, in this program:

```
procedure main {
1.      if (1 == 1) then {
2.              a = a;
        } else {
3.              b = 1;
        }
}
```

First, the visitor will visit the "main" procedure. The list of direct statements in the procedure is obtained from the AST. In this case, the procedure has only 1 direct statement, which is statement 1. An edge is drawn from the "main" procedure to statement 1.

When processing statement 1, the list of statements in the "then" and the "else" condition is obtained from the AST, which in this case is statement 2 and 3. Then, an edge is drawn from statement 1 to 2, and 1 to 3. This is because any variables used in statement 2 and 3 will be used by statement 1 as well.

When processing statement 2, since it is a basic assignment statement, an edge is drawn from statement 2 to all variables that are on the right-hand side of the assignment, which in this case is the variable "a".

When processing statement 3, since it is an assignment variable with nothing on the right-hand side, no edges are added.

Hence, Figure 3.4.1.6.1 below shows the graph obtained which is passed into `ConnectivityChecker`.

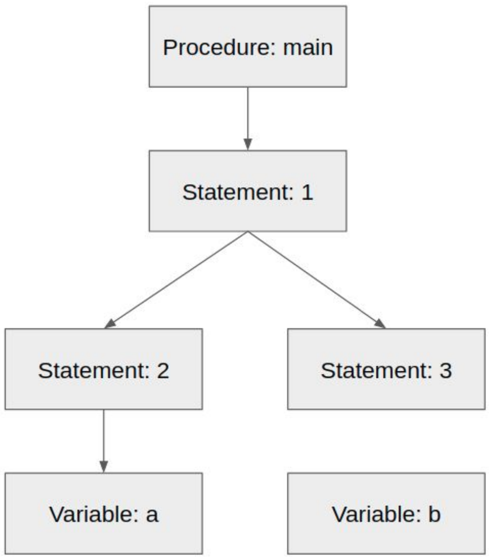


Figure 3.4.1.6.1: Graph Obtained for Uses Relationship

The `ConnectivityChecker` would then find the transitive closure of this graph. To check if either a statement or a procedure uses a variable, one can simply check if the target variable is reachable via a series of edges in the graph, which can be done by querying the `ConnectivityChecker`.

For example, Uses(1,”a”) is true because statement 1 can reach the variable “a” via the intermediate node, statement 2.

The only data structure stored is the mapping between procedure names/variable names to `ConnectivityChecker` graph node index using `std::map<ProcName, int>` and `std::map<VarName, int>`. Statement numbers use their values as the index.

3.4.1.7 PKBModifies

This class extracts all Modifies relationships between entities. The only major difference to PKBUses is that the variables in the left side of assignment statements are checked, as opposed to checking the right side. The details are symmetric to that of PKBUses and omitted for brevity.

3.4.1.8 PKBPattern

This class will answer queries regarding patterns. **Assignment pattern matching** has two parts:

1) Matching LHS variable
2) (Partially) Matching RHS expression

The first part is simple enough; simply consult the PKBEntityList class to fetch the assignment statement AST node and verify the LHS variable is equal to the one in the pattern.

The second part is done by (sub)string matching. This is possible because the canonical string representation (including enclosing brackets for each variable, constant and binary expression, with consistent whitespace rules) of expressions is a perfect hash for the expression. Moreover, sub-expressions’ string representations will be sub-strings of the string representation of expressions it is contained in. In this class, we take all assignment statements and convert its RHS expressions to strings and store it in a map. To perform the second check, we do either an exact string match or a substring match depending on whether partial matches are allowed.

As an illustration of the hashing process, consider the following assignment statement.

```
5.    ...
6.    x = a + ((6 / b) * (42069 - c));
7.    ...
```

The canonical string representation for the right-hand-side expression is as follows:

“((a)+(((6)/(b))*((42069)-(c))))”

The brackets are coloured based on bracket nesting level. It is then clear how a sub-expression can be matched. We similarly hash a subexpression to be matched and perform a substring matching. For instance:

```
a) a(_, _"6 / b"_)          ⇒     "((6)/(b))"        // Is a substring
b) a(_, _"a"_)              ⇒     "(a)"              // Is a substring
c) a(_, _"b * 42069"_)      ⇒     "((b)*(42069))"    // Not a substring
```

The case for full expression match is then performing a full string matching.

In this way, pre-computation reduces a SIMPLE programming language expression matching problem in runtime to a (sub)string matching problem in runtime.

The data structure used to hash all the possible (sub)expressions is `std::unordered_set<std::string>`.

For While and If pattern matching, we could store a map of while/if statement numbers to the set of variables in its conditional expression, and when we try to match a non-wildcard pattern, we do find on the set for the singular variable.

For example, these will be the set stored for these expressions:

```
a) if (a + b < a + 100 + c)  ⇒     {"a", "b", "c"}
b) while(var < 1)            ⇒     {"var"}
```

And these would be the variable to find for these queries:

```
a) ifs("b", _, _)            ⇒     "b"          // Find string
b) while("a", _)             ⇒     "a"          // Find string
```

We pre-compute this map from the statement numbers to sets.

The data structure used to store the map is `std::unordered_map<StmtNo, std::unordered_set<VarName>>`.

3.4.1.9 PKBCalls

This class extracts all Calls(*) relationships between procedures. This is done by filling two adjacency matrices for two directed graphs, one for direct Calls relationships and another for transitive Calls* relationships. These graphs are stored in an auxiliary class `ConnectivityChecker`. It first extracts an edge list of all direct Calls relationships from the AST, adding an edge for each call statement. `ConnectivityChecker` accepts this edge list as input to its constructor to generate the adjacency matrix for direct Calls relationships, and then subsequently uses the Floyd Warshall algorithm to compute the adjacency matrix for indirect calls relationships.

Checking whether two procedures fulfil a Calls(*) relationship is then a matter of consulting the appropriate adjacency matrix to verify there exists an edge.

This is very symmetric to the case of PKBParent (the above paragraphs were copy-pasted from the PKBParent section, substituting “Parent” for “Calls”). Hence, we omit a detailed walkthrough of the algorithm and only show how the transitive Calls* relations are stored for the sample program in figure 3.4.1.9.1 below.

```
procedure main {
1.    call p1;
}
procedure p1 {
2.    call p2;
}
procedure p2 {
3.    call p3;
}
}procedure p3 {
4.    call p2;
}
```

	main	p1	p2	p3
main	0	1	1	1
p1	0	0	1	1
p2	0	0	1	1
p3	0	0	1	1

Figure 3.4.1.9.1: Mapping between Procedure Names

The only data structure stored is the mapping between procedure names to `ConnectivityChecker` graph node index using `std::map<ProcName, int>`.

This is also where semantic validation of the SIMPLE program is done. Checking for calls to non-existent procedures can be caught naturally when drawing the edges, while recursive and cyclic calls can be checked for after the `ConnectivityChecker` is filled:

```
for each procedure p:
      if is_calls_s(p, p):
            throw “Recursive or cyclic call detected”
```

3.4.1.10 PKBNext

This class extracts all Next(*) relationships between statements. It allows querying for Next and Next* relationship between any two statements in O(1) time with O(N) preprocessing. This is done in three steps:

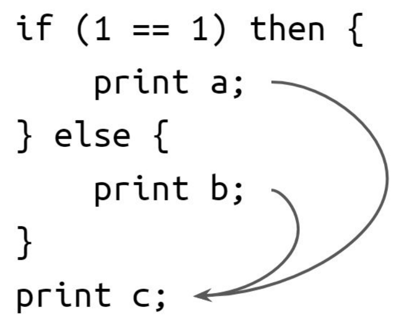
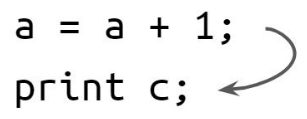
Step 1: Obtaining the Control Flow Graph (CFG)

Each statement represents a node in the graph, and the edges refer to all possible one-hop execution steps between statements in the program.

First, we would use the visitor pattern to find all statement lists in the program. This is because two consecutive statements in a statement list would contain an execution step.

Suppose statement A and statement B are consecutive statements in a statement list. Then, there are two cases to consider: either statement A is an “if statement”, or it is not an “if statement”. If statement A is an “if statement”, then two new edges are added: one from the last statement in the “then” part of statement A to statement B, and one from the last statement in the “else” part of statement A to statement B. Otherwise, if statement A is not an if statement, then a new edge is simply added from statement A to statement B.

Case where statement A is an “if statement”	**Case where statement A is not an “if statement”**

 `if (1 == 1) then {`
`    print a;`
`} else {`
`    print b;`
`}`
`print c;`
	 `a = a + 1;`
	`print c;`

Next, we would add the edges for the container statements (ie the if and while statements). For the if statements, an edge is added from the if statement to the first statement in the “then” and the “else”. For the while statements, an edge is added from the while statement to the first statement inside the container.

Case with “if” container	**Case with “while” container**
`if (1 == 1) then {` `asd = 123;` `print a;` `} else {` `print b;` `}`	`while (1 == 1) {` `print a;` `}`

These cases cover all the possible execution steps in the program.

For example, this simple program would produce following CFG given in figure *3.4.1.10.1*:

```
procedure main {
1.     while (1 == 1) {
2.            a = 1;
3.            b = 2;
       }
4.     if (1 == 1) then {
5.            a = 3;
       } else {
6.            b = 4;
       }
7.     c = 3;
}
```

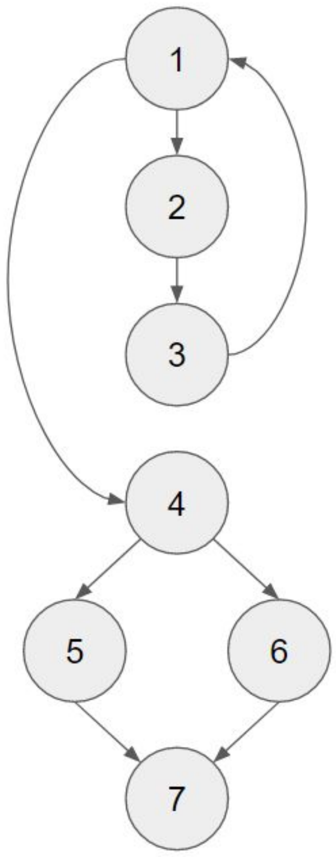


Figure 3.4.1.10.1: CFG for SIMPLE Program

This reachability graph is used to check for Next relations. To allow the checking of Next* relations, more precomputation is needed, which will be explained in the next steps.

Step 2: Compressing strongly connected components

Note that the resulting CFG graph is possibly cyclic. Hence, we compress the graph and merge strongly connected nodes together, to create a new compressed graph that is acyclic. To compress the strongly connected components, tarjan's algorithm is used (Complexity O(V+E)).

Figure 3.4.1.10.2 below shows the new compressed graph from the CFG:

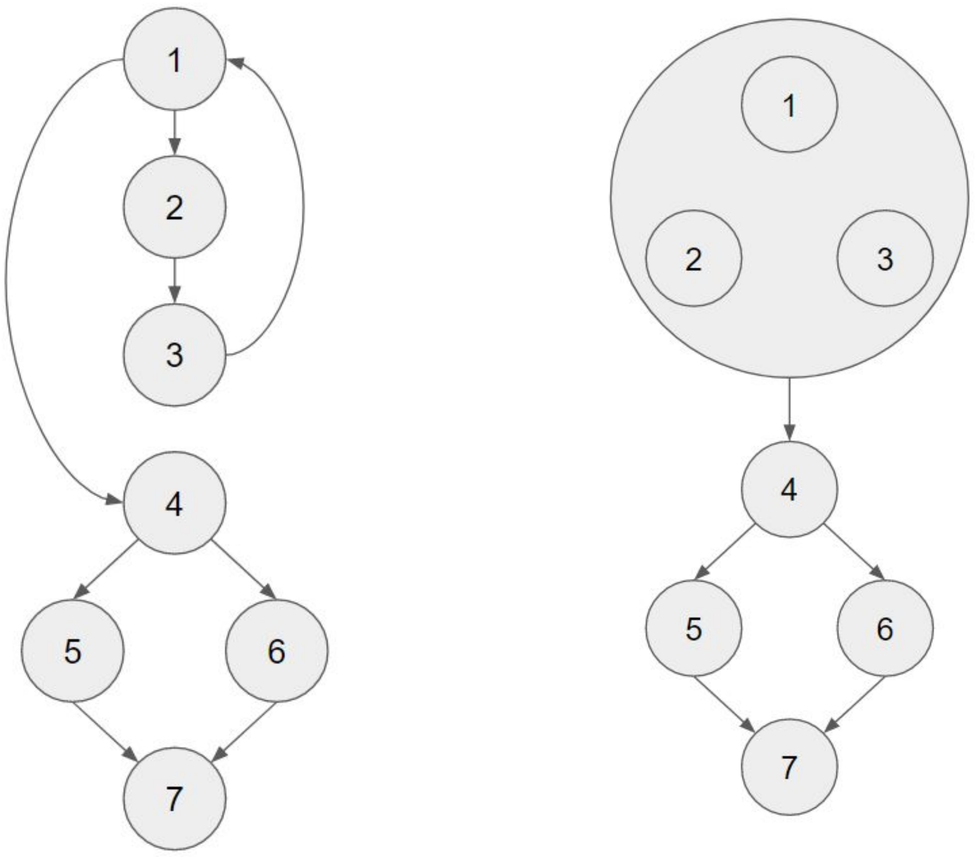


Figure 3.4.1.10.2: Compressed Graph from CFG

Step 3: Run precomputation on planar graph

The last step is crucial for PKBNext to compute Next* reachability between any two statements in constant time.

We notice that the compressed acyclic graph is planer and also satisfies the necessary conditions for Kameda's algorithm which is described in Appendix C. Kameda's algorithm allows us to query if a component can reach another component in the graph. Hence, we run that algorithm to precompute and store meta-data about each strongly connected component.

Using Kameda's algorithm, we would label each component with a pair of values, which is computed by running DFS from left-to-right and right-to-left, and the graph above would produce the following pairs as labels shown in Figure 3.4.1.10.3 below:

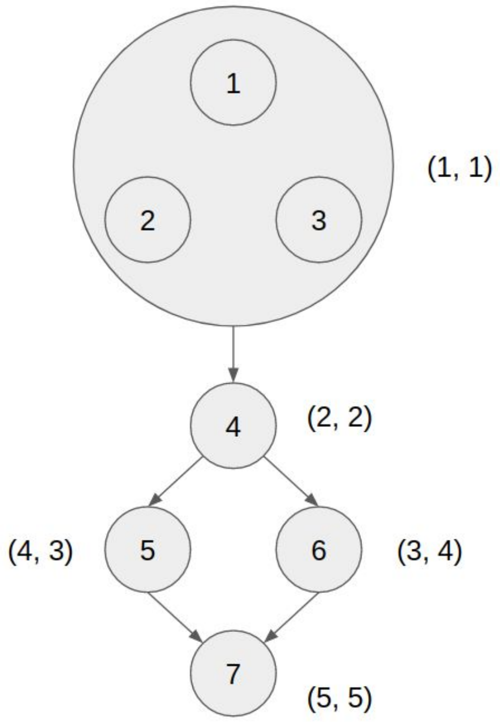


Figure 3.4.1.10.3: Mapping Between Procedure Names

To check if the component containing nodes {1,2,3} can reach the component containing node 6, we will check if each value is the pair is smaller than the values in the other pair, which in this case, we check if (1, 1) ≤ (3, 4), and since it is smaller, node 6 is reachable. On the other hand, node 5 cannot reach node 6 because (4, 3) is not ≤ (3, 4).

Hence, when querying Next*(a, b), firstly, we check if a and b are in the same component. If they are in the same component, and the component size is greater than 1, then Next*(a,b) is true. Else, if a and b are not in the same component, we can query Kameda's algorithm if the component of a can reach the component of b.

In this way, we satisfy the requirement of not storing tables of Next(*) relations in tables, and instead store a linear amount of data in the CFG in linear time, and we can support queries in O(1) time.

There are multiple data structures used to store the required data. The list of next statements are stored in `std::vector<std::vector<StmtNo>>` to map from the statement number (the index) to the list of next statements. To store the Strongly Connected Components, we use `std::vector<int>` to map the statement number as the index to the ID of the component. The reachability of the components are stored in the auxiliary class `KamedaReachability`.

3.4.1.11 PKBAffects

This class will answer queries regarding Affects, and it takes in PKBEntityList, PKBModifies, PKBUses, as well as PKBNext, and would use them as blackboxes for computation.

Since the definition of Affects uses the control flow graph, hence, PKBNext is used to provide the control flow graph. Each statement between the two assignments also needs to be checked for Modifies and Uses, and this is done using PKBModies and PKBUses respectively.

Given a particular assignment statement, a DFS (depth first search) approach is used to find all assignment statements that are affected by that particular assignment statement. This is done by going through all the control paths (provided by PKBNext) and stopping if we encounter a Read/Assign/Call statement that modifies the variable in the first assignment statement. To further prune the paths to check, we only check paths that are reachable to the target statement (checking reachability can be done in O(1) using PKBNext's Next* function).

Suppose in the following SIMPLE program given in Figure 3.4.1.11.1 below, we would like to check Affects(1, 7). Hence, we would like to find all statements that are reachable from statement 1 in a path that does not modify variable x, and that statement must be able to reach statement 7. We call a statement "valid" if it does not modify variable x and is able to reach statement 7, and "invalid" otherwise.

```
procedure main {
1.      x = 1;
2.      if (1 == 1) then {
3.              if (1 == 1) then {
4.                      x = 2;
5.                      y = 3;
                } else {
6.                      y = 1;
                }
7.              y = x;
        } else {
8.              x = 2;
9.              x = 3;
        }
}
```

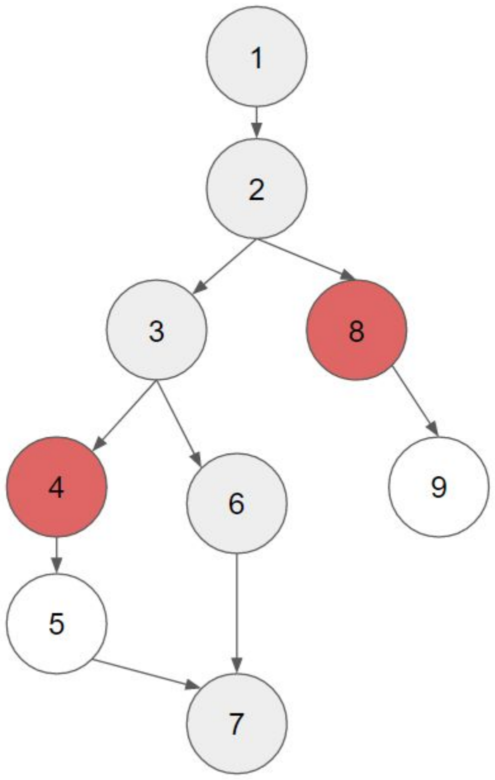


Figure 3.4.1.11.1: Simple Program and Corresponding CFG

In the diagram above, red statements are “invalid” statements in the control path, and white statements are statements to skip.

Statement 4 modifies variable x, hence, that is an “invalid” statement. Hence, statement 5 does not need to be checked. Statement 8 cannot reach statement 7, hence, it is also “invalid”, and statement 9 does not need to be checked.

Finally, among all the grey statements, we can check all assignments inside which Uses the variable x, and those would be the statements Affected by statement 1.

Since we can get all relevant statements that statement 1 Affects, we could repeat this process to find all transitive relations for Affects* via another DFS, treating the Affects relation as edges in the graph.

Figure 3.4.1.11.2 below shows a simulation of the Affects* DFS, where it is assumed that it can get the list of neighbours (statements it affects) as a blackbox.

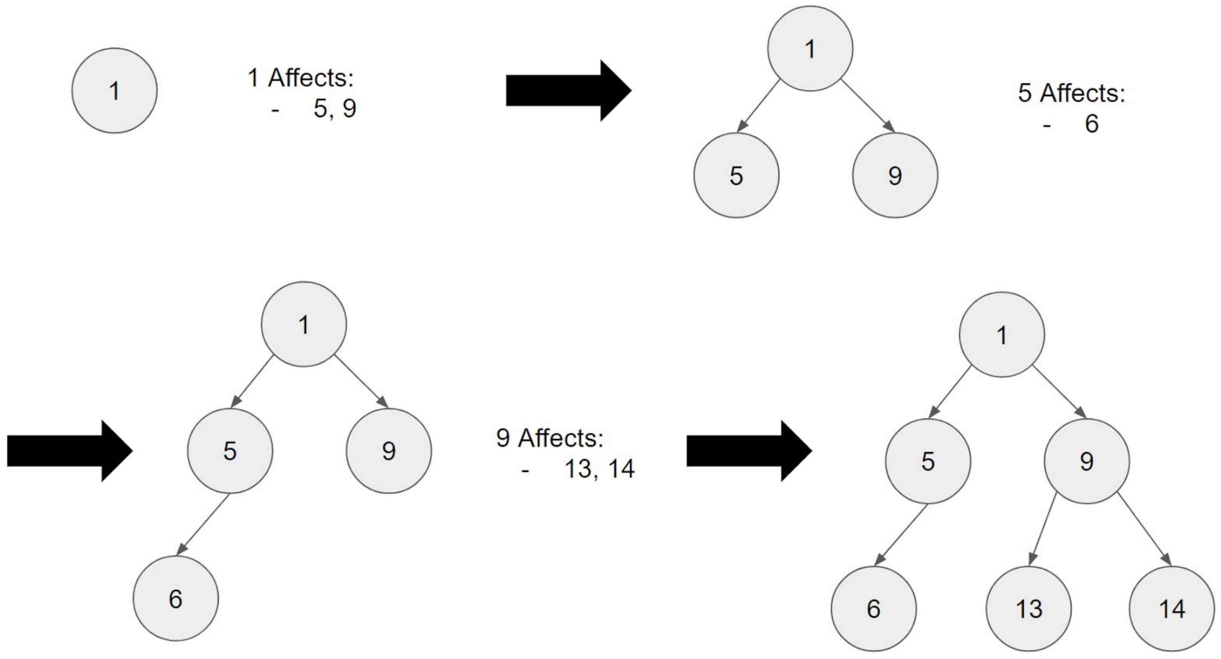


Figure 3.4.1.11.2: Simulation of Affects DFS*

Here, we would like to find statements which is Affected* by statement 1. We check who are the neighbours of statement 1 (who statement 1 Affects), which are statements 5 and 9. Then, we check which statements are Affects by statement 5, which is statement 6. We repeat this process until no new statements can be visited. All nodes reachable would be Affected* by statement 1.

DFS takes up to O(number of nodes + number of edges), and since the number of edges is O(N) (where N is the number of statements), hence, finding all statements Affected by a particular statement would take O(N). Hence, checking for Affects relation takes O(N) per query in the worst case.

For Affects*, finding the neighbours of each node takes O(N) (since it needs to find all statements Affected. Hence, checking for Affects* takes O(N^2) per query in the worst case.

However, the time complexity above is for the worst case. If the statements are near each other, and there are very few nodes in the path between the statements, then it would take faster, as some nodes could be pruned by ignoring “invalid” nodes.

3.4.2 Design Considerations: PKB

The two main concerns are ease of comprehensive testing and ease for extension.

The PKB concerns itself with the extractions of various design abstractions; we need to be sure that these extractions are done correctly. Therefore sub-components (and API) should have clear Separation of Concerns to allow for easy testing.

The PKB's functionality also evolves throughout the various iterations, and perhaps in unexpected ways depending on the extensions implemented. The design should therefore be open for extension.

3.4.3 Design Decision: PKB Class API

3.4.3.1 Options Available

General Routing API

In designing the API, one consideration is the volatility of API as we move through the iterations of the development. For instance, if we decided to expose one retrieval function per design entity/abstraction, then the PKB API must be augmented each time we introduce a new design abstraction. Therefore, we wanted the PKB API to be as small and general as possible. An example of a PKB API exposed to the Query Processor is this:

```
bool test_relation(Relationship rs, DesignEntity de1, DesignEntity de2);
```

Where `Relationship` is an enum which tells the PKB which design abstraction is being requested (Modifies/Parent/etc.) and uses this to route the `test_relation` request to the appropriate algorithm class.

Specific API

An alternative design (also mentioned above) was to have specific PKB API's per design abstraction. For instance, testing the Follows relation between two entities could look like this:

```
bool test_follows(DesignEntity de1, DesignEntity de2);
```

While testing the Uses relation between a statement and a variable could look like this:

```
bool test_uses_statement(DesignEntity de1, DesignEntity de2);
```

Notice how this differs from our current design; the Query Processor (or any component that uses the PKB Class API) will have to be explicit which function to call, rather than call a general "routing" function where the decision as to where exactly to "route" to is determined by an enum parameter.

3.4.3.2 Evaluation Criteria

The most important criteria in determining the design of the PKB Class API is the ease with which it can be extended to add support for new query relationships. This aids us in keeping the PKB extendible beyond Iteration 1, saving us much work for Iterations 2 and 3. Another criterion is the ease with which we can test the PKB. Individual methods in the PKB API should be testable without too much set-up. Finally, we should also consider the ease of use of the API. Ideally, users of the API (in the case of this project, the Query Processor component) should be given methods that allow them to make more general calls to the API.

3.4.3.3 Evaluation of Options

	Design considerations		
	Ease for extension	**Ease of testing**	**Ease of use**
General Routing API	✓	✓	✓
Specific API		✓	

3.4.3.4 Final Decision and Rationale

The specific API design makes the PKB API more volatile, needing change every time we introduce a new abstraction. The benefit, however, is that because we **don't** have general functions, the specific API can specify exactly what types of arguments to accept. So Instead of `DesignEntity` for both arguments in `test_uses`, we could specify that the first argument must be a statement number, and the second number a variable. This adds one layer of checking, in that the caller is forced to be type-correct to use the API.

However, our current design of the Query Processor handles data uniformly; all design entities are strings, reasons implicit in how we designed our query tables. The benefits of having specific functions cannot be realised without explicit conversions between strings and the required argument types. This makes the API more cumbersome to use, and we compromise ease of use of the API for extra safety.

On the other hand, the general API design creates a PKB that is **open for extension**, as the `Relationship` enum mentioned in the example above contains all design abstractions in SPA, even those that have not been implemented yet (e.g. Next/Affects). Additional bonus abstractions can also be added simply by adding fields to the `Relationship` enum, without affecting user code. The other methods in the API to retrieve whole tables of relations are similarly implemented. This makes the PKB API much less volatile, thus making this the final choice for the design of the PKB Class API.

3.4.4 Design Decision: PKB Sub-components

3.4.4.1 Options Available

Modular PKB

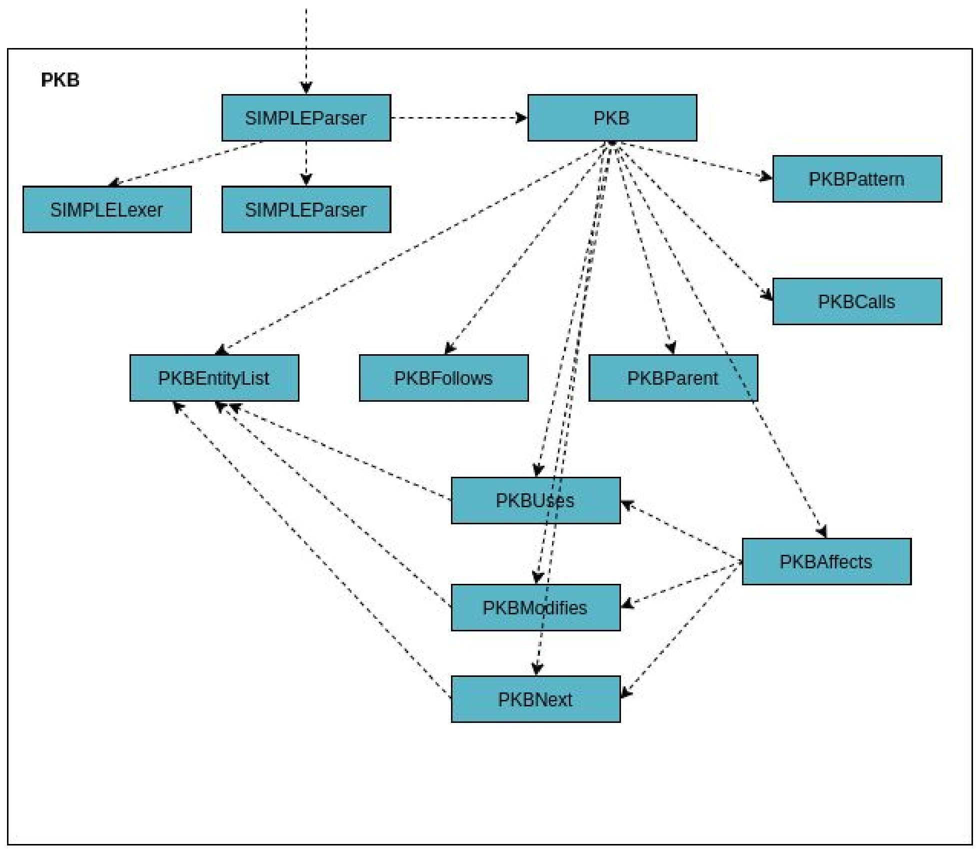


Figure 3.4.4.1.1: Component-classes of PKB. Adding design abstractions (e.g. Next, Affects) is akin to adding more subclasses

As mentioned in the SPA architecture, the SIMPLEParser is a PKB sub-component that is "static". To reiterate, this component is decoupled strongly from the rest of the PKB sub-components and offers a single method to parse a program string into an AST.

There are also other components of the PKB. The main PKB class contains instantiations of other PKB classes; one for each design abstraction. Since each of the design abstractions has its own algorithms and optimisations, the PKB component consists of one "algorithm class" per abstraction. Even the design entities extraction is separated into its own class. These classes are decoupled from each other as much as possible; only if an algorithm class requires the results of another class (most often the class for extracting design entities) will it have access to this class.

Each of these algorithm classes are responsible for extracting their design abstractions from the AST. Each of these classes also offers internal API to the PKB class (and possibly to each other) to retrieve these design abstractions. For instance, the API exposed by the PKBFollows class is of the form:

```
bool is_follows(<EntityType1>, <EntityType2>)
bool is_follows_s(<EntityType1>, <EntityType2>)
```

The Pattern class is also similar, and tests pattern matching for single statements:

```
bool is_assign_pattern_matching(<StmtNo>, <Pattern>)
```

In all cases, if the arguments supplied do not match the required design entity type for the relationship/pattern, the function returns false.

Monolithic PKB

An alternative design was to merge the algorithm classes of the PKB together to form a monolithic component containing all the different entities and abstractions in a single class.

3.4.4.2 Evaluation Criteria

There are a few key criteria that can be used to evaluate the options available. Firstly, we can consider the optimisation opportunities available - how much we can optimise algorithms for specific relationships in the PKB without affecting the rest of the algorithms. Next, we should consider whether the design leaves the PKB open for extension. This would mean that new relationships can be added to and handled by the PKB with ease. Another criterion would be the ease with which we can test the PKB sub-components. Ideally, each component should be testable on its own without having to stub the entire PKB. Finally, we can consider whether the PKB is closed for use by other modules.

3.4.4.3 Evaluation of Options

	Design considerations			
	Optimisation opportunities	**Open for extension**	**Ease of testing**	**Closed**
Modular PKB	✓	✓	✓	✓ (until fundamentally new forms of questions need to be answered)
Monolithic PKB	✓	✓		

3.4.4.4 Final Decision and Rationale

The benefits of the monolithic PKB design would be that the computation of a specific design abstraction could use information about another related design abstraction. For instance, checking if an assignment statement matches the pattern a(“x”, _) is equivalent to checking Modifies(a, x). However, this complicates testing, as the QueryProcessor ideally wants to mock an instance of the PKB. By using a modular PKB and limiting the fields and interface to a simpler facade, it becomes both easier to test the correctness of extracting and retrieving each design abstraction, and easier to use for testing. Furthermore, as the PKB grows to support more design abstractions, this will affect the interface, which makes a monolithic PKB’s interface **not closed** for use by other modules.

3.5 Query Processor Component

The Query Processor component is made up of the following classes:

- QueryProcessor
- QueryParser
- QueryBuilder
- QueryValidator
- Query
- QueryClause
- QueryTable
- QueryEvaluator
- QueryClauseGrouper
- ACConstraintGraph
- ConstraintGraphOptimizer
- ResultsProjector

Figure 3.5.1 below shows the structure of the Query Processor Component and how the classes interact with one another.

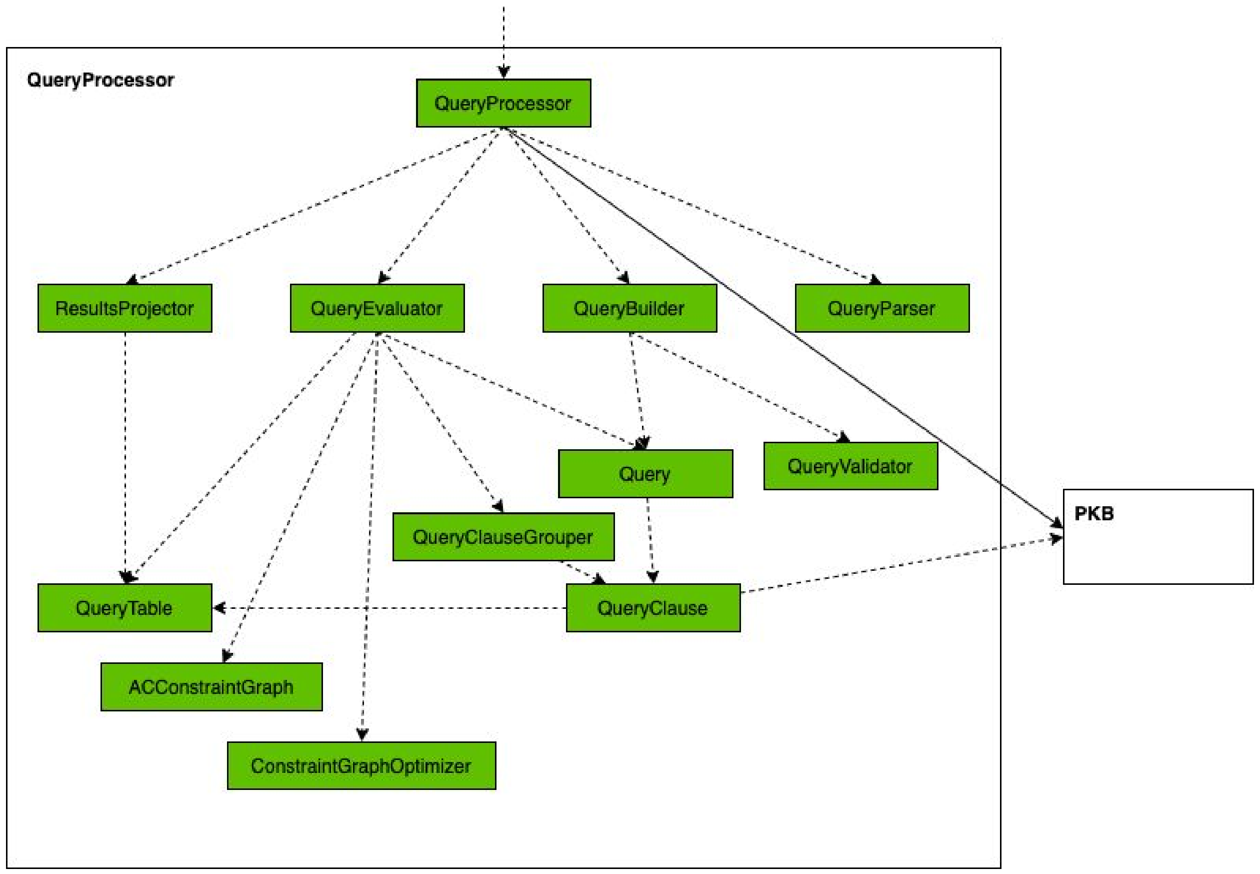


Figure 3.5.1: Structure of Query Processor Component

The sequence diagram in Figure 3.5.2 below shows a high-level view of how a query is processed in the Query Processor component.

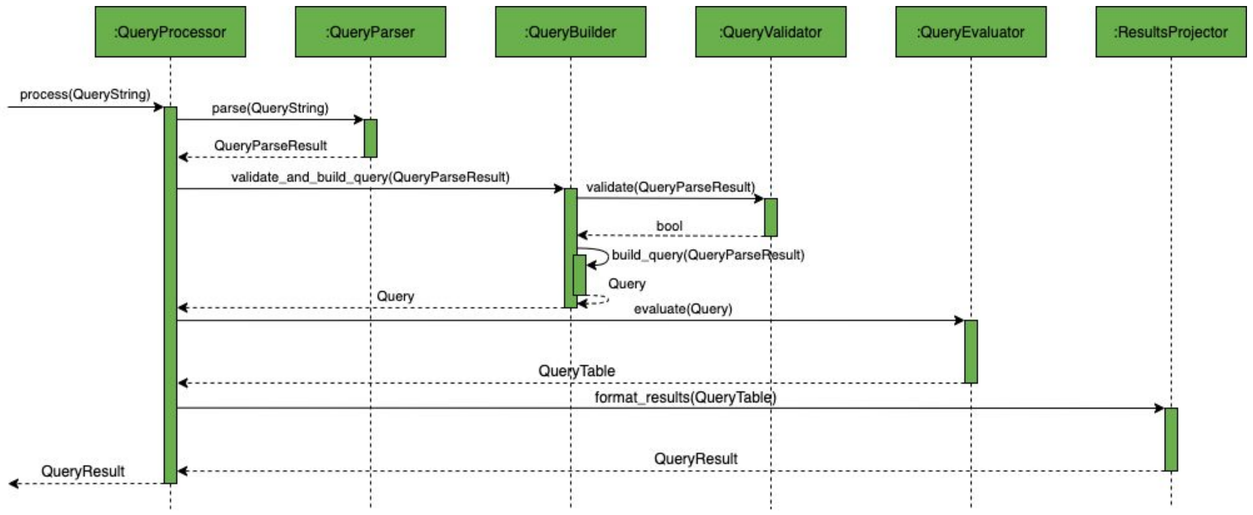


Figure 3.5.2: Sequence Diagram on How a Query is Processed in Query Processor

3.5.1 Query Processor Sub-components

In the following sections, we describe each of the Query Processor sub-components in detail.

3.5.1.1 QueryProcessor

This class serves an abstraction to be used by the UI for query processing and evaluation, by calling the different functions of the subclasses in the QueryProcessor and returning the final result.

The following describes the roles of the sub-classes in the QueryProcessor:

- **QueryParser**: validates the syntactic validity and parses user queries written in Program Query Language (PQL) into a QueryParseResult
- **QueryBuilder**: builds a Query using the QueryParseResult
- **QueryValidator**: validates the semantic validity of the QueryParseResult before the QueryBuilder uses the QueryParseResult to build a Query
- **Query**: internal representation of a user query that is easy to use for evaluation
- **QueryClause**: represents a such that, with or pattern clause, and acts as a facade for calls to the PKB to retrieve a QueryTable that represents this clause
- **QueryTable**: represents a table which the QueryEvaluator can operate on to produce the final result
- **QueryEvaluator**: evaluates the Query, producing a final result QueryTable
- **QueryClauseGrouper**: arranges and groups the QueryClauses from a Query to optimize evaluation time
- **ACConstraintGraph**: represents a constraint satisfaction problem (CSP) to be solved with the arc consistency algorithm (AC3) (more details in 3.5.1.8 QueryEvaluator and 3.5.1.10 ACConstraintGraph)
- **ConstraintGraphOptimizer**: static class that takes in a ACConstraintGraph and returns an optimized ACConstraintGraph
- **ResultsProjector**: formats the final result QueryTable into a QueryResult (an unordered set of strings) for printing

3.5.1.2 QueryParser

This subcomponent performs syntactic analysis on the given PQL and returns a syntactically validated QueryParseResult. A representation of QueryParse result can be seen in Figure 3.5.1.2.1 below.

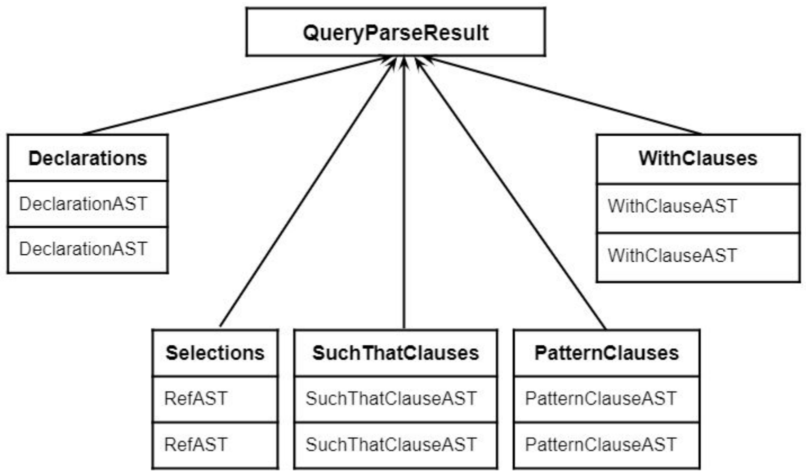


Figure 3.5.1.2.1: Representation of Query Parse Result

In order to simplify reasoning, we split lexical analysis and syntactic analysis into two components, the lexer and the parser. The lexer converts the PQL string (a list of characters) into a list of tokens, and the parser converts the list of tokens into a QueryParseResult.

The lexer identifies contiguous sequences of letters/digits (names), contiguous sequences of digits (integers), symbols, etc. while discarding all whitespace. This is symmetric with the SIMPLELexer in Section 3.4.1.1, with contextual (grammar) changes.

The parser is a standard parser written using parser combinators, in a fashion that closely mirrors the grammar as described in the SPA requirements. This is symmetric with the SIMPLEParser in Section 3.4.1.1, with contextual (grammar) changes.

Let's use the following PQL query string as an example:

```
procedure p; variable v;
Select p such that Uses(p, v)
```

The QueryLexer produces the following list of tokens after lexing through the above query string.

Tokens															
procedure	p	;	variable	v	;	Select	p	such	that	Uses	(	p	,	v	)

With the list of tokens, the QueryParser performs syntactic validation and constructs a QueryParseResult for semantic validation by the QueryValidator. (Refer to Section 3.4.1.1 for how syntactic validation is performed.)

After the tokenized query has been validated for syntax errors, the QueryParseResult is constructed by building a list of declarations, a list of selected references, a list of such that clauses, a list of pattern clauses and a list of with clauses.

The following is how the QueryParseResult looks like after parsing:

QueryParseResult

DeclarationASTList

Type	Names
procedure	p
variable	v

SelectedRefASTList

Type	Contents	Attr
SYNONYM	p	-

Relationship	LHS		RHS	
	Type	Contents	Type	Contents
Uses	SYNONYM	p	SYNONYM	v

3.5.1.3 QueryValidator

This class performs semantic validation on the QueryParseResult by breaking down the QueryParseResult into declarations, selected synonyms/attributes, such that clauses, pattern clauses and with clauses and validating them one after the other. Since syntactic validation is performed by the QueryParser, we can safely assume that only syntactically valid queries make it to the stage.

We perform validation of such that and pattern clauses using a series of “if” statements based on the relationship and synonym respectively. If any of the validation steps produce an error, the entire query is deemed to be semantically invalid and false is returned, otherwise true is returned.

Here is an example of how semantic validation is performed in a sequential order.

Let’s use the following PQL query string as an example and assume it has been converted to a QueryParseResult by the QueryParser:

```
procedure p; variable v; assign a;
Select p.procName such that Uses(p, v) pattern a(v, _) with a.stmt# = 3
```

Declarations
Firstly, the QueryValidator will validate the DeclarationASTList by checking that each synonym-design entity pair is valid. It will also check that each synonym is used only once.

DeclarationASTList

Type	Names
procedure	p
variable	v
assign	a

As procedure p, variable v and assign a are all valid synonym-design entity pairs and synonyms are not reused, the DeclarationASTList passes validation.

Examples of invalid declarations:

- Invalid synonym-design entity pair: `proc p;`
- Repeated synonym: `procedure p; print p;`

Selected Synonyms/Attributes

Next, the QueryValidator will validate the SelectedRefList:

- If synonym is selected
  - Check that synonym has been declared
- If attribute is selected
  - Check that synonym has been declared
  - Check that synonym-attribute pair is valid

SelectedRefASTList

Type	Contents	Attr
ATTRIBUTE	p	procName

As p.procName is a valid synonym-attribute pair, the SelectedRefASTList passes validation.

Example of invalid selected references: (p declared as procedure)

- Invalid synonym-attribute pair: `p.stmt#`

Such That Clauses

Next, the QueryValidator will validate the SuchThatClauseASTList by determining whether the type-contents pairs are valid on both the left-hand side and right-hand side based on the type of relationship used.

SuchThatClauseASTList

Relationship	LHS		RHS	
	Type	Contents	Type	Contents
Uses	SYNONYM	p	SYNONYM	v

The above relationship is valid as a procedure can Uses a variable.

Example of invalid such that clauses: (v, v1, v2 declared as variable)

- Invalid Follows: `Follows(v1, v2)` as Follows is only valid for statement references
- Invalid Uses: `Uses(_, v)` as it is unclear whether '_' stands for a statement or a procedure

Pattern Clauses

Next, the QueryValidator will validate the PatternClauseASTList by determining whether the pattern synonym is valid. Valid pattern synonym is assign, while or if. It will also check whether the type-contents pair on the left-hand side is valid. Valid LHS is synonym (variable), IDENT (name) or _.

PatternClauseASTList

pat_synonym	target	is_exact_match	optional<pattern>	num_args
a	Type | Contents SYNONYM | v	false	null	2

The above pattern clause is valid since the pattern synonym is an assign synonym, and the left-hand side argument is a variable synonym.

Example of invalid pattern clauses: (p declared as procedure)

- Invalid pattern synonym: `pattern p(v, _)`
- Invalid LHS argument: `pattern a(p, _)`

With Clauses
Finally, the QueryValidator will validate the WithClauseASTList by determining that the left-hand side and right-hand side arguments are the same type (string/integer). If synonym-attribute pairs are used, they will also be validated.

WithClauseASTList

LHS			RHS		
Type	Contents	Attr	Type	Contents	Attr
ATTRIBUTE	a	stmt#	INTEGER	3	-

The above with clause is valid since a.stmt# (assign) is valid, and both left-hand side and right-hand side arguments map to integers.

Example of invalid pattern clauses: (a declared as assign)
- Invalid attribute: `with a.value = 3`
- Mismatch LHS and RHS types: `with a.stmt# = “three”`

Once all the above steps are performed, “true” is returned to the QueryBuilder and it will proceed to build the Query object using the valid QueryParseResult.

3.5.1.4 Query
A query contains a list of selected references and a list of query clauses. Each query clause can represent such that, with and pattern clauses. Figure 3.5.1.4.1 below shows the general structure of a Query object.

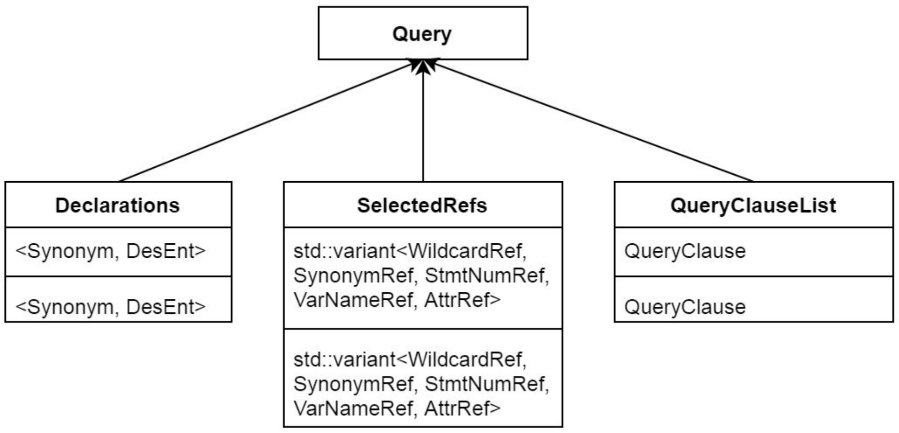
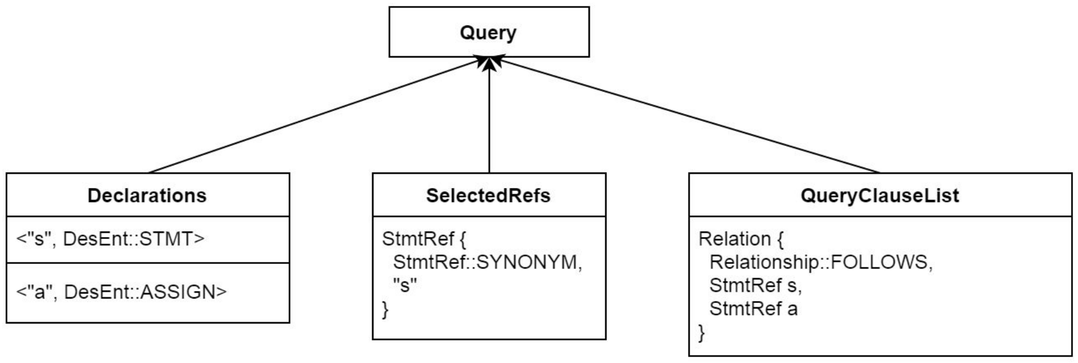

Query
Declarations
<Synonym, DesEnt>
<Synonym, DesEnt>
SelectedRefs
std::variant<WildcardRef, SynonymRef, StmtNumRef, VarNameRef, AttrRef>
std::variant<WildcardRef, SynonymRef, StmtNumRef, VarNameRef, AttrRef>
QueryClauseList
QueryClause
QueryClause

Figure 3.5.1.4.1: Representation of Query Object

As an example, the following query

```
stmt s; assign a;
Select s such that Follows(s, a)
```

would produce the Query object as seen in Figure 3.5.1.4.2 below:



Query
Declarations
<"s", DesEnt::STMT>
<"a", DesEnt::ASSIGN>
SelectedRefs
StmtRef { StmtRef::SYNONYM, "s" }
QueryClauseList
Relation { Relationship::FOLLOWS, StmtRef s, StmtRef a }

Figure 3.5.1.4.2: Representation of Example Query Object

3.5.1.5 QueryBuilder

The QueryBuilder class is responsible for building a Query object using the QueryParseResult for evaluation by the QueryEvaluator by exposing a single method:

```
Query validate_and_build_query(QueryParseResult &qast)
```

First, validate the QueryParseResult by calling the validate method in the QueryValidator. If it is valid, proceed to build the query, if not, we throw an error that the query is semantically invalid. After the validation is done, the query building process will be guaranteed not to throw any errors.

Next, we begin building the Query object. The following table describes the building process for each attribute of a Query.

Attribute	**Description**
Declarations	Iterate through the list of DeclarationAST in the QueryParseResult and construct a map for where synonyms are mapped to the design entity they represent based on each DeclarationAST.
SelectedRefs	Iterate through the list of RefAST in the QueryParseResult and construct a list of selected references (references can either be a statement, entity or attribute reference) based on each RefAST.
QueryClauseList	Iterate through the lists of SelectedSynonym, SuchThatClause, PatternClause and WithClause and construct a list of QueryClause. A QueryClause is able to represent selected, such that, pattern and with clauses).

The built Query is then passed to the QueryEvaluator for evaluation.

3.5.1.6 QueryClause

A query clause can represent a such that, with or pattern clause. It acts as a facade for calls to the PKB API, so that such calls can be moved out of the Query Evaluator class. To facilitate the evaluation of queries, QueryClause exposes the `get_table` method to retrieve a corresponding QueryTable from the PKB:

```
QueryTable get_table(PKB pkb)
```

All calls to the PKB API are made within `get_table`; the exact methods called depend on the type of clause, and the arguments passed to the PKB methods depend on the arguments of the clauses themselves. To illustrate, the execution of `get_table` for each type of clause is described below with an example clause for each.

Such-That Clause

In the case of such-that clauses, `get_table` will call `filter_satisfying_relation` from the PKB API:

```
DesEntDataTable filter_satisfying_relation(Relationship rs,  DesEntDataDomain dedd1,
DesEntDataDomain dedd2)
```

Take the clause `Follows(1, s1)` as an example.

- The Relationship passed will be an enum value, `Relationship::FOLLOWS`.
- The left and right domains are vectors that hold all possible values of the corresponding left and right clause arguments.
    - In this case, the left clause argument is the constant `1`, so the left domain will be a vector containing `1`.
    - The right clause argument is the synonym `s1`; the list of declarations are checked and, if for the sake of example we assume it refers to all statements, the right domain will be a vector containing all statement numbers in the SIMPLE program.

`filter_satisfying_relation` will return a `DesEntDataTable`, which is a one or two-column table of design entity data entries in row-major order. In the case of `Follows(1, s1)`, the table will contain a single column as only a single synonym, `s1`, is present in the clause. `s1` will be the column's header and the column itself will contain all statement numbers in the SIMPLE program *except* the first statement, since `Follows(1, 1)` would evaluate to false. For example, if the program had 99 statements in it:

s1
2
...
99

`get_table` will convert this DesEntDataTable into a QueryTable and return it to QueryEvaluator. A detailed explanation on the QueryEvaluator can be found in Section 3.5.1.8.

Pattern Clause

The method exposed by the PKB for pattern clauses is:

```
DesEntDataTable filter_satisfying_pattern(Pattern pat, SPAPattern spat,
DesEntDataDomain dedd1, DesEntDataDomain dedd2)
```

For example, given the clause `pattern a (_, “y”)`:

- The Pattern passed will be an enum value, `Pattern::ASSIGN`.
- The SPAPattern passed will be the SPAPattern stored in the pattern QueryClause by the QueryBuilder while the Query object was being created (refer to 3.5.1.5 QueryBuilder for details).
- The left and right domains are vectors that hold all possible values of the pattern type and all possible values of the pattern clause arguments.
    - The pattern clause is for matching assignments, so a vector of every assignment statement number is passed.
    - The right argument is a wildcard, so a vector of every statement number is passed.

`filter_satisfying_pattern` will return a `DesEntDataTable`, which is a one or two-column table of design entity data entries in row-major order. In the case of `pattern a (_, “y”)`, the table will contain a single column with the header `a` and containing all assignment statement numbers where the right-hand side of the assignment has a match to “y”.

With Clause

The method exposed by the PKB for with clauses is:

```
DesEntDataTable filter_satisfying_with(
        DesEnt left_type, DesEnt right_type,
        Attr left_attr, Attr right_attr,
        DesEntDataDomain left_entities, DesEntDataDomain right_entities)
```

Taking the clause `with s.stmt# = c.value` as an example:

- `left_type` is the type of `s` as an enum value `DesignEntity::STMT`
- `right_type` is the type of `c` as an enum value `DesignEntity::CONSTANT`
- `left_attr` is the type of `s.stmt#` as an enum value `Attr::STMTNO`
- `right_attr` is the type of `c.value` as an enum value `Attr::VALUE`
- `left_entities` is a vector of all statement numbers i.e. all possible values of `s`
- `right_entities` is a vector of all constant values i.e. all possible values of `c`

`filter_satisfying_with` will return a `DesEntDataTable`, which is a one or two-column table of design entity data entries in row-major order. In the case of `with s.stmt# = c.value`, the table will contain two columns with the headers `s` and `c` and the contents being the values of `s` and `c` that equal each other.

3.5.1.7 QueryTable

The sole responsibility of the QueryTable class is to store the data entries and headers; no table operations are provided by this class except for data iteration, table creation, and table equality comparison. Operations such as the joining of tables are provided by other classes such as AbstractJoinAlgorithm.

The headers are stored as a vector of synonyms while the data entries are stored as a vector of DesignEntityData in flattened row-major order. The rationale behind this design decision is discussed fully in Section 3.5.4 Design Decision: Query Table.

Since the data entries are stored in flattened row-major order, we also wrote a VecStrideIterator class in order to easily iterate over columns or rows. This class allows us to iterate over a stride of the data: instead of taking every consecutive element, we take every element separated by a given offset. This was accomplished by overriding the operators of the standard C++ vector iterator. In this way, despite the data entries being in flattened row-major format, we can create a VecStrideIterator that allows other classes and methods to iterate over the table's column entries with little difficulty.

For the creation of QueryTables, this class exposes three methods, each of which takes different arguments:

```
QueryTable from_rows(Headers headers, vector<Row> rows)

QueryTable from_references_rows(vector<Reference> references, vector<Row> rows)

QueryTable from_cols(Headers headers, vector<Column> columns)
```

In the cases where headers are passed to the QueryTable, we can simply take the headers as is and use the data entries in the rows/columns to fill the table in flattened row-major order.

However, in the case where we are passed a vector of references instead, these references are converted to synonyms first before being stored as table headers. The reference is checked for what type of reference it is (since a Reference is, in fact, a C++ variant of a WildcardRef, SynonymRef, AttrRef, StmtNumRef, or VarNameRef) and, if it is a SynonymRef or AttrRef, converted to a synonym and stored as a header.

3.5.1.8 QueryEvaluator

This class is responsible for evaluating a Query object and returning a final result QueryTable containing only the selected synonyms and their possible values given the clauses in the query. To that end, it exposes a method, `evaluate`, that takes the Query object from the QueryProcessor class and returns the result QueryTable:

```
QueryTable evaluate(Query query, PKB pkb)
```

In order to evaluate a query, QueryEvaluator takes two main steps.

1. The query's clauses are arranged such that the more restrictive clauses are evaluated first. More details on the exact order in which the clauses are grouped and arranged can be found in Section 3.5.1.9 QueryClauseGrouper, but the main purpose of this step is to optimize the actual evaluation of the clauses.
2. The query, having had its clauses arranged to optimize evaluation time, is evaluated using the Arc Consistency Algorithm #3 (AC3). This algorithm reduces the possible values each synonym in the query can take until only the values that satisfy all the query's clauses remain. More details on how this algorithm and how it is used to evaluate a query can be found in Appendix B - Arc Consistency Algorithm.

The two steps above cover the bulk of the work done in evaluating a query. As might be guessed from their descriptions, most of their logic is abstracted away into other sub-components (the details of which can be found in subsequent sub-component sections). The grouping of clauses is done in the QueryClauseGrouper class, **then AC3 is used on each group in separate instances of ACConstraintGraph classes**. As such, the QueryEvaluator class itself is mainly concerned with handling the output of those classes so that a final QueryTable can be returned.

Before proceeding further, it is ***highly recommended*** that one first read through Appendix B - Arc Consistency Algorithm in order to obtain a general understanding of how our program uses AC3 to evaluate a query. With that understanding, we can proceed with a description of the exact steps taken to evaluate a query by the QueryEvaluator.

First, the QueryClauses in the `query` are separated into different QueryClauseLists by the QueryClauseGrouper component. This is done through the following static method exposed by the QueryClauseGrouper class:

```
vector<QueryClauseList> QueryClauseGrouper::group_clauses(QueryClauseList clauses)
```

The `clauses` passed to `group_clauses` are extracted from the `query`.

For each clause in each of the lists of QueryClauses, QueryEvaluator calls the following method exposed by the QueryClause class:

```
QueryTable get_table(PKB pkb)
```

This method retrieves a QueryTable created from the respective QueryClause. Because the QueryClause is an abstract class that represents every type of clause, the QueryEvaluator does not have to consider the type of clause at all. Furthermore, this abstraction allows the QueryEvaluator to be decoupled from the PKB; no calls to the PKB take place in QueryEvaluator. More details on how the QueryClause class itself handles the different clause types and how it calls the PKB can be found in Section 3.5.1.6 QueryClause. Regardless, the retrieved QueryTable is placed in a list for evaluation.

At this point, QueryEvaluator begins the actual evaluation of the query using AC3. To start executing the algorithm, QueryEvaluator calls the constructor of the ACConstraintGraph class, passing it the list of QueryTables and set of selected synonyms:

```
ACConstraintGraph (vector<QueryTable> tables, set<Synonym> seleceted_synonyms)
```

The ACConstraintGraph instance returned by the above method contains a constraint graph whose domains have already been reduced to their smallest possible sizes.

All that remains is to construct and return the final QueryTable. For each of the ACConstraintGraph instances (for each query group), the table with selected synonyms as the headers and the synonyms' corresponding domains as the columns is retrieved. The final QueryTable is then constructed by taking the cartesian product of all these tables. With that, the final QueryTable is returned to the caller, QueryProcessor.

3.5.1.9 QueryClauseGrouper

This class optimises query evaluation by grouping certain clauses together. To that end, it exposes a static method for the QueryEvaluator to call:

```
vector<QueryClauseList> QueryClauseGrouper::group_clauses(QueryClauseList clauses)
```

This method takes in a list of clauses, separates the clauses into different lists, and returns the separated lists to the QueryEvaluator. In addition, we sort the groups in the following priority:

1. Groups without synonyms
2. Groups with synonyms

Since clauses without synonyms can be evaluated almost immediately, they are placed at the front so that if any of these clauses evaluate to false, evaluation can be terminated early and immediately.

The grouping of clauses is done as described in lecture. We want each group of clauses to be transitively connected to each other via shared synonyms.

For example, the following Query:

```
stmt s; assign a1, a2, a3; procedure p;

Select s such that Follows (1, 2) and Parent (s, a3) and Parent (s, 3) and Calls (p,
“main”) and Modifies(s, a1) and Affects (a1, a2) pattern a1 (v, _)
```

will have its clauses grouped like so:

Group 1	**Clauses without synonyms**			`Follows (1, 2)`
Group 2	**Clauses with synonyms**	**Clauses without selected synonyms**	**Connected synonym** `p`	`Calls (p, “main”)`

Group 3			**Connected synonyms** `a1, s`	`Affects (a1, a2)`
				`Pattern a1(v, _)`
				`Modifies (s, a1)`
				`Parent (s, a3)`
				`Parent (s, 3)`

To demonstrate how our query clause grouping is transitive, in Group 3, we colour-code the important shared-synonyms that result in the grouping. In particular, each clause belongs in the group because it shares either **`a1`** or **`s`** with another clause.

In Group 2, the “Calls” clause has only 1 synonym “p” that is not shared with any other clause. Hence it belongs in its own group.

In Group 1, there are no synonyms for the “Follows” clause. If there were any other clauses without synonyms, they would also form their own group.

When grouping clauses in this way, we can then treat the original query as evaluating three separate sub-queries, then taking the cartesian product of their individual results:

```
Select s such that and Parent (s, a3) and Parent (s, 3) and Modifies(s, a1) and
Affects (a1, a2) pattern a1 (v, _)
Select BOOLEAN such that Calls (p, “main”)
Select BOOLEAN such that Follows(1, 2)
```

3.5.1.10 ACConstraintGraph

As mentioned in Section 3.5.1.8 QueryEvaluator, our program uses the arc consistency algorithm (AC3) to solve a query, which it treats as a constraint satisfaction problem (CSP). More details on this algorithm and how we use it to solve a query can be found in Appendix B - Arc Consistency Algorithm. This class, ACConstraintGraph represents the CSP to be solved with AC3 for a query. **The following explanations rely upon understanding the algorithm, and only discusses implementation of the algorithm. Reading Appendix B is mandatory before reading this section.**

ACConstraintGraph represents the CSP by defining the domains as a map `m_domains`, where each synonym points to an AC3Domain (unordered set of possible values the synonym can take), and defining the current status of the CSP (the constraint graph) as a map `m_constraint_graph`, where each synonym points to another map with synonyms as keys and AC3Constraints (unordered set of pairs, representing a row in the table backing the constraint) as values.

- `using AC3Domain = unordered_set<Synonym>`
- `using AC3Constraint = unordered_map<DesEntData, unordered_set<DesEntData>>`
- `unordered_map<Synonym, AC3Domain> m_domains`
- `unordered_map<Synonym, unordered_map<Synonym, AC3Constraint>> m_constraint_graph`

Note that in our code, AC3Domain and AC3Constraint do not exist and are flattened inside the ACConstraintGraph for performance reasons. Actual data-representation in the code is also not so simple as the above, as we use C++ smart pointers to support copy-on-write. We use these terms in the report only for clarity and illustration purposes.

To solve the CSP, ACConstraintGraph also contains the following methods (listed in their order of use):

- `ACConstraintGraph (vector<QueryTable> tables, set<Synonym> seleceted_synonyms)`
- `bool arc_consistency(unordered_set<Synonym> syns_to_update)`
- `bool arc_consistency(unordered_set<pair<Synonym, Synonym>> worklist)`

The flow of execution is illustrated in Figure 3.5.1.10.1 and described in greater detail below:

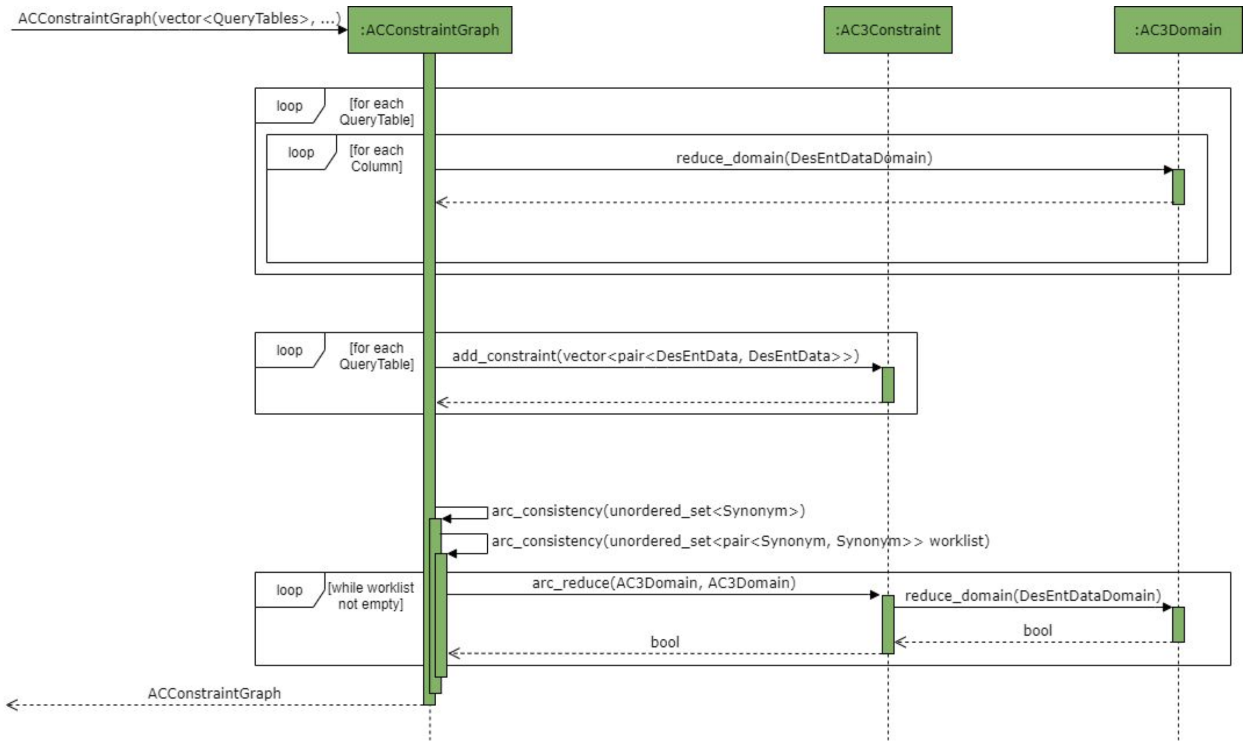

ACConstraintGraph(vector<QueryTables>, ...)
:ACConstraintGraph
:AC3Constraint
:AC3Domain
loop [for each QueryTable]
loop [for each Column]
reduce_domain(DesEntDataDomain)
loop [for each QueryTable]
add_constraint(vector<pair<DesEntData, DesEntData>>)
arc_consistency(unordered_set<Synonym>)
arc_consistency(unordered_set<pair<Synonym, Synonym>> worklist)
loop [while worklist not empty]
arc_reduce(AC3Domain, AC3Domain)
reduce_domain(DesEntDataDomain)
bool
bool
ACConstraintGraph

Figure 3.5.1.10.1: Sequence Diagram for AC3 Algorithm

The constructor for ACConstraintGraph is the entry point to the algorithm. It is called by the QueryEvaluator class and takes a list of QueryTables from which the domains and constraints of the CSP are created, and the set of selected synonyms. The ACConstraintGraph constructor then uses the tables to fill in `m_domains` and `m_constraint_graph`.

The AC3Domains are first updated to only have values that appear in all input query tables where the synonym is a header. This is intuitive, since for a particular synonym, it can only possibly take on values that appear in all QueryTables where it is present.

The AC3Constraints are then updated. Although named “constraint”, this class is a whitelist of permissible pairwise assignments between two different synonyms. Therefore, for a particular AC3Constraint between synonym “a” and “b”, it is the intersection of all (two-column) QueryTable’s where both “a” and “b” both appear.

Lastly, the `arc_consistency` method is called. This performs the Arc Consistency Algorithm to iteratively reduce domains based on the AC3Constraint whitelist, until no more domains can be reduced.

It starts with needing to check all domains at least once via `arc_consistency(unordered_set<Synonym> syns_to_update)`. This method does nothing but loop through all neighbour synonyms of a particular synonym, before passing the list to `arc_consistency(unordered_set<pair<Synonym, Synonym>> worklist)`, the main workhouse of the algorithm.

For each ordered pair of Synonyms <a, b> in the worklist, we attempt to reduce the domain of one of the Synonyms with respect to the AC3Constraint tying them together (say “b”). If the domain is reduced, that implies that neighbours of “b” (including “a”) might be reduced as a result too. We then add those pairs <b, *> to the worklist, and iterate.

After this is done, the ACConstraintGraph would have minimized the domain sizes with respect to all constraints (clauses).

To retrieve the actual result, we bruteforce and check the cartesian product of all reduced domains against all constraints. If for a specified assignment of all used synonyms (a row in the cartesian product), it satisfies all constraints, then we add it to the output table.

Because domain sizes are much smaller than they originally were, bruteforce checking is much faster with less rows to check for. Note that if there is a domain that is empty, it implies that there is NO valid assignment of all used synonyms. We skip the enumeration of results and return an empty table.

3.5.1.11 ConstraintGraphOptimizer

To further minimize the number of rows to bruteforce and check, we can use optimisations specific to the AC graph. **These optimisations are run on the ACConstraintGraph where each domain is non-empty** (as mentioned in the previous section, if any domain is empty then the resulting table is empty).

To illustrate the process of graph optimization, consider the following ACConstraintGraph where a vertex (synonym) is highlighted if it is selected:

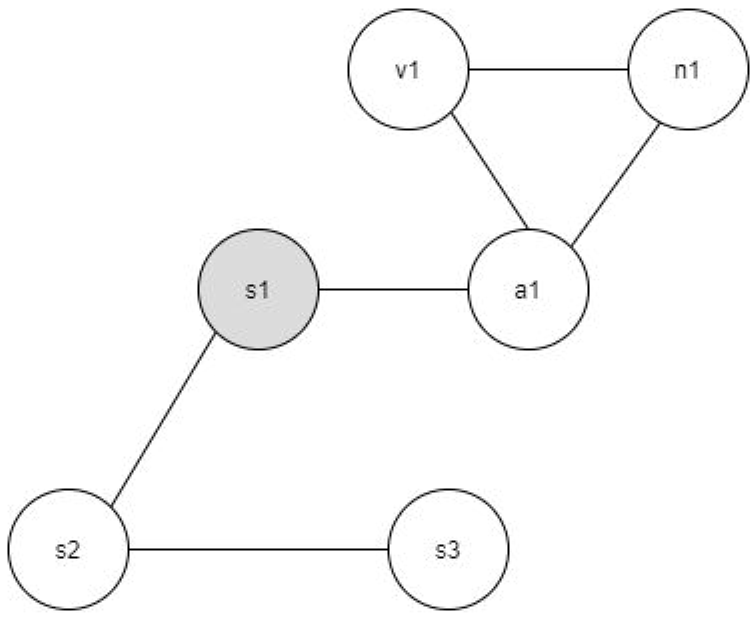


Figure 3.5.1.11.1: Initial Unoptimized Constraint Graph

How the ACConstraintGraph in Figure 3.5.1.11.1 above is obtained (what clause(s) correspond to each constraint) is unimportant for purposes of the optimization. We only require that the domains are all non-empty, and the graph is arc-consistent.

We perform many iterations of two optimization strategies until no further optimizations are possible. The first step is Tree-Pruning.

Tree-Pruning

A vertex (synonym) can be removed along with its edges if the following conditions holds:

1) Synonym has one or zero related constraints, i.e. its graph degree is 1 or 0
2) Synonym is unselected

In the given example above, the vertex “s3” can be removed without consequence. A proof sketch is as follows: Assume there is an assignment everywhere else (for “s1”, “s2”, “a1”, “v1”, “n1”), i.e. `<s1: x ,s2: y, a1: a, v1: b, n1: c>`. Then there must be an assignment for “s3” with respect with this assignment for the other synonyms, since arc-consistency means that regardless of what value “s2” takes (this case `y`) there exists a value for “s3” (say `<s3: z>`) that satisfies the constraint between “s2” and “s3”. Since this is the only constraint “s3” is involved in, we can remove the “s3” vertex and the resulting graph is equivalent with respect to selected synonyms.

We remove vertex “s3” and its constraint to get the following graph in Figure 3.5.1.11.2 below:

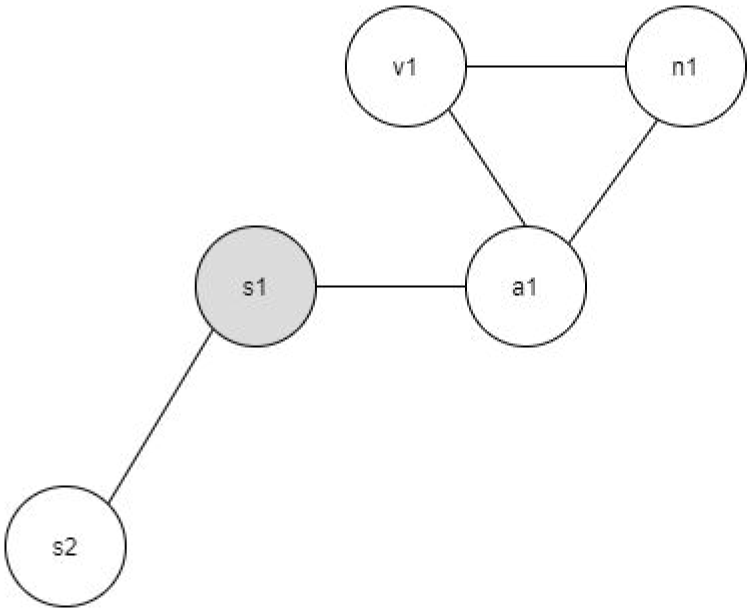


Figure 3.5.1.11.2: Constraint Graph with Vertex “s3” Removed

Once again, we can remove “s2” since the degree of “s2” is now 1 as seen in Figure 3.5.1.11.3 below:

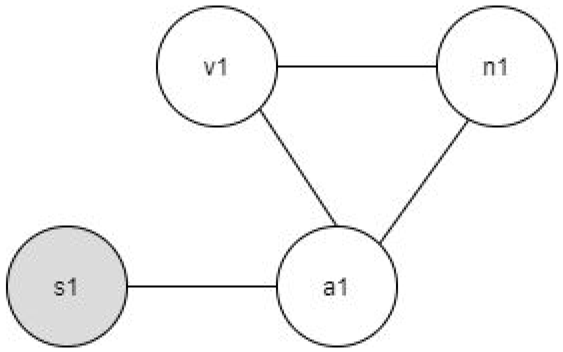


Figure 3.5.1.11.3: Constraint Graph with Vertex “s2” Removed

Our final ACConstraintGraph now only has four domains. For a more concrete example, refer to the example given in Appendix B. Vertices “w”, “c” and “a” can all be recursively removed, which explains why all 16 rows in the cartesian product are all valid, and in reality we only need to output the domain of “s”.

The next step of optimization is Chain-Reduction.

Chain-Reduction

To further minimize the number of rows to bruteforce, we can remove unselected synonyms (vertices) if they meet the following condition:

1) Synonym has exactly two related constraints, i.e. its graph degree is 2
2) Synonym is unselected

This reduces the number of domains in the cartesian product when enumerating results.

We continue from the previous example to illustrate. Previously, we mentioned that AC3Constraint's are whitelists of valid pairwise assignments, i.e. tables. So the intuition is this, to remove a synonym (vertex) with two constraints (degree 2), we merge the two constraints (table join) to create one new one (the resulting table has 3 columns, but since the vertex is unselected we can remove it and project the other two columns).

In the above, the first chain to reduce is the `v1-n1-a1` chain where we remove vertex "n1". Since there already is an existing constraint between "v1" and "a1", we have to update the constraint between "v1" and "a1", effectively doing a table intersection. The resulting graph is seen in Figure 3.5.1.11.4 below, where the red edge represents an updated constraint:

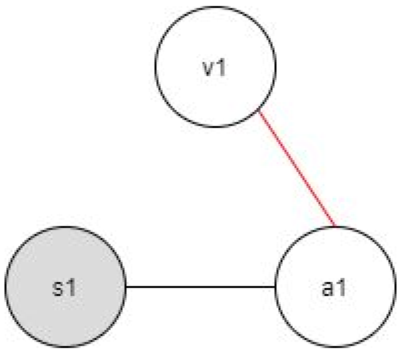


Figure 3.5.1.11.4: Constraint Graph with Vertex "n1" Removed

We do it again for "a1", and get the following graph in Figure 3.5.1.11.5 below:

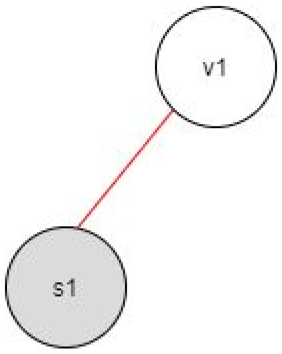


Figure 3.5.1.11.5: Constraint Graph with Vertex "a1" Removed

This time, there was no existing constraint between the end-points of the chain (“s1”, “v1”), so the red edge represents a new constraint as a result of the merging of constraints.

At this point, we can no longer do Chain-Reduction. However, the new graph allows for one more iteration of Tree-Pruning! We remove vertex “v1”.

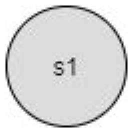


Figure 3.5.1.11.6: Constraint Graph with Vertex “v1” Removed

Finally, our graph has only one domain as seen in Figure 3.5.1.11.6 above. As it turns out, there is no need for cartesian products in enumerating results, we simply output the domain of “s1” as the final result.

3.5.1.12 ResultsProjector

This class formats the final result QueryTable into a QueryResult (an unordered set of strings) for printing. It takes the query’s declarations, query’s select targets, and a QueryTable containing the desired results. Using the declarations and select targets, it extracts the targeted column from the QueryTable, formats the column’s entries into a QueryResult, and returns the QueryResult.

3.5.2 Design Considerations: Query Processor

The functionalities of the Query Processor component are abstracted away and are only exposed via the QueryProcessor class, allowing us to adhere to the information-hiding principle of software engineering.

The Query Processor flow is also split up into components to follow the Single Responsibility Principle. Specifically, the `process()` method is now an organising method, a scaffold that creates the general shape of the behaviour but then delegates the work to other methods in other components. Thus, to fully discuss the design considerations of the Query Processor Component, the design of each sub-component is discussed in its own sub-section below.

3.5.3 Design Decision: QueryBuilder

This section describes our design decision for the storage of query clauses.

3.5.3.1 Options Available

Store such that, with and pattern clauses in separate lists

Each type of query clause would be stored in separate lists. In this case, there would be no need to create an overarching abstract data type that could represent every type of query clause.

Store such that, with and pattern clauses in a single list

Store all query clauses (such that, pattern, with) in a single list by creating an abstract data type, QueryClause.

3.5.3.2 Evaluation Criteria

There are two main criteria to consider when choosing between the options listed above. Firstly, we can consider the ease of implementation; the easier an option is to implement, the more appealing it might be. The second criteria to consider is how well the option supports optimisation of the query evaluation process. Both criteria are important, but greater weight can be placed on the second criteria since time is a limiting factor when evaluating queries.

3.5.3.3 Evaluation of Options

	Design considerations	
	Easy to Implement	**Optimisation Support**
Store such that and pattern clauses in separate lists	✓	
Store such that and pattern clauses in single list		✓

3.5.3.4 Final Decision and Rationale

We chose to go with the second option for Iterations 2 and 3, compared to the first option in Iterations 1. This is because storing all query clauses in a single list allows us to compare query clauses and sort them regardless of their more specific type. This would then give us a greater ability to optimize the evaluation of queries since query clauses can be sorted such that joining them would create intermediary table(s) with fewer rows.

For Iterations 2 and 3, we felt this greater ability to optimize would be more important than ease of implementation, as we now have to deal with queries that contain more than 1 such that and 1 pattern clause as in Iteration 1. As a bonus, the Query object is now much simpler, since it only contains three fields (list of declarations, list of selected synonyms, list of query clauses) compared to the Query object from before which contained a different list for each type of query clause.

3.5.4 Design Decision: QueryTable

This section describes our design decision for storing data in QueryTable.

3.5.4.1 Options Available

Store the data in a 2D vector (`vector<vector<DesEntData>> contents`)

In this way, rows were easily accessible but columns could only be accessed through each row. Implementation was simple as we only had to maintain 1 variable when creating and modifying the QueryTable. We ran into difficulty when we wanted to operate on whole columns by removing them from the QueryTable when merging the cross product. The process of doing so on the 2D vector would result in unnecessarily complicated code.

Store data in separate row and column vectors (`vector<DesEntData> Row and vector<DesEntData> Column`)

This implementation made it easier to perform operations such as cross product, as compared to using a 2D vector.

Store data in flattened row-major (`vector<DesEntData> contents`)

In this case, all entries in the table would be stored in a single vector. To make it easy to iterate through both rows and columns, we could create a new iterator class that inherits from the standard C++ vector iterator. This new iterator could take a numerical offset value so that each consecutive element it iterates through is separated from the previous element by said offset. For example, with a vector `{a, b, c}` and an offset of `1`, the iterator would only consider `{a, c}`.

3.5.4.2 Evaluation Criteria

There are two main criteria to consider when evaluating the options above. Firstly, how easy it is to implement the options above should be a consideration, especially since the choice of table format will affect the methods needed to create and read from the tables. Secondly, the ease with which we can perform operations (such as joining) tables should be considered; the more complicated and difficult it is to implement such operations, the more likely it is for mistakes to be made that could affect query evaluation results further down the line.

3.5.4.3 Evaluation of Options

	Design considerations	
	Easy to Implement	**Easy to Perform Operations**
Store data in 2D vector	✓	
Store data in row and column vectors		✓
Store data in flattened row-major	✓	✓

3.5.4.4 Final Decision and Rationale

As of Iteration 2, we have chosen to use a flattened row-major format to store our table contents. It provides an easier way to iterate over rows and columns. Most importantly, it is the most cache-friendly design option when we consider the iterations on the table for query evaluation. This is because the overhead of a single vector is much less compared to storing the table as a vector of vectors, which would be required if we were to store it in normal row-major or column-major order.

We chose to go with the last option as it would be the easiest to write performant code which supports only 1 type of table. We would not have to change the format or support alternative formats. As we are starting to write more complex functions to support query optimisation, this added benefit would make writing new code more straightforward and compatible.

3.5.5 Design Decision: QueryEvaluator

This section describes our design decision for the evaluation of query clauses, and how said evaluation can be optimized.

3.5.5.1 Options Available

Allow QueryEvaluator to call PKB

Initially, we left the bulk of the whole evaluation process to QueryEvaluator. Evaluation would start in the `evaluate` function, where it would receive the whole Query object and PKB pointer. Then, it would split Query into its key components - selected synonym, such that clauses and pattern clauses. Finally, it would make calls to the PKB to get the respective QueryTable for each clause before merging all tables to get a final one, where the selected synonym will be taken from.

Delegate PKB calls to QueryClause

The responsibility of the QueryEvaluator for this option will simply be to call the `get_table` method for each QueryClause. The QueryClause will then call the PKB itself. The QueryEvaluator also no longer has to consider the type of each QueryClause as it retrieves the QueryTables. In this way, the QueryClause acts as a facade between the QueryEvaluator and PKB and the QueryEvaluator is decoupled from the PKB.

A sequence diagram on how calls to PKB are delegated to QueryClause for query evaluation is shown in Figure 3.5.5.1.1 below.

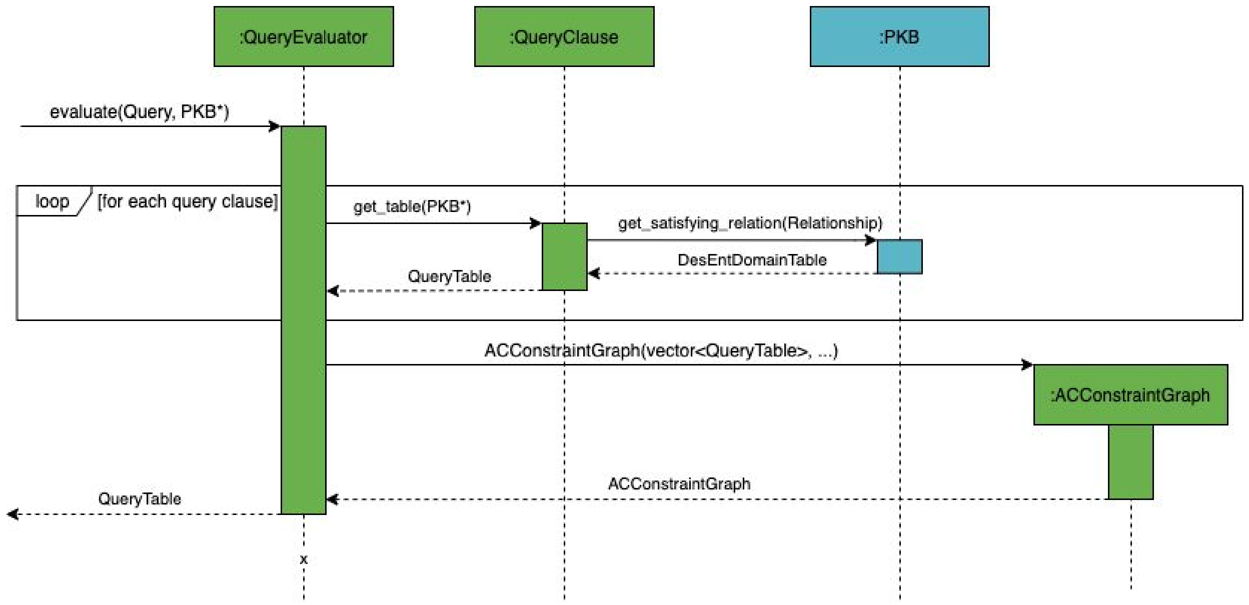


Figure 3.5.5.1.1: Sequence Diagram for Evaluation of Query

3.5.5.2 Evaluation Criteria

For the options above, we can use two criteria to choose between them. The first is the ease of implementation. The easier it is to implement, the less likely we are to make mistakes that cause query evaluation to fail. The second criterion is how well we can adhere to the Separation of Concerns. This is especially important for the Query Evaluator sub-component as it has the potential to expand in scope and complexity very quickly due to the number of responsibilities it could potentially shoulder. With that in mind, this second criteria should almost certainly be given greater weight over the first.

3.5.5.3 Evaluation of Options

	Design considerations	
	Easy to Implement	**Separation of Concerns**
Allow QueryEvaluator to call the PKB	✓	
Delegate the calls to PKB to QueryClause		✓

3.5.5.4 Final Decision and Rationale

For Iterations 2 and 3, we chose to move all calls to the PKB to the QueryClause class in order to better adhere to the Single Responsibility Principle. Although the first design is intuitive, procedural, and easier to implement, modifying it to allow for query optimisation required in Iteration 3 would mean excessive coupling of QueryEvaluator with many other components as well as giving QueryEvaluator an excessive number of responsibilities. These would increase coupling and violate the Single Responsibility Principle and Separation of Concerns respectively.

3.5.6 Design Decision: Query Optimisation

This section details our design decision for query optimisation where we were choosing between using a QueryClauseSorter or the AC3 algorithm.

3.5.6.1 Options Available

QueryClauseSorter

The QueryClauseSorter class aims to optimise queries by sorting them in a way such that clauses that require more time and resources to evaluate will be prioritised last. In this way, if the earlier clauses turn out to be false, query evaluation can be terminated, and the later clauses which require more resources do not have to be evaluated.

Sorting would be done within each group of query clauses, where each group was formed by the QueryClauseGrouper. A detailed breakdown of the QueryClauseGrouper can be found in Section 3.5.1.9. After which, the sorted, grouped queries would be evaluated by the QueryEvaluator.

The sorting order is listed below in descending order of priority:

1. With clauses
2. Such that clauses (can be precomputed)
    a. One synonym
    b. More than one synonym
3. Pattern clauses
4. Such that clauses (cannot be precomputed)
    a. One synonym
    b. More than one synonym

According to the project requirements, the clauses that can and cannot be precomputed are listed accordingly in the table below:

Can be Precomputed	Cannot be Precomputed
• Follows/Follows* • Parent/Parent* • Uses • Modifies • Calls/Calls* • Next	• Next* • Affects/Affects*

For example, the following Query:

```
stmt s; assign a, a1; variable v;

Select BOOLEAN such that Follows (a, 2) and Parent (s, a) with a.stmt# = 4 and
Affects (a, a1) and Next* (a, 6) Pattern a (v, _)
```

which will be grouped together by QueryClauseGrouper as they have a connected synonym (`assign a`) will be sorted in the following order:

1	**With clauses**		`a.stmt# = 4`
2	**Such that clauses (can be precomputed)**	**One synonym**	`Follows (a, 2)`
		More than one synonym	`Parent (s, a)`
3	**Pattern clauses**		`pattern a (v, _)`
4	**Such that clauses (cannot be precomputed)**	**One synonym**	`Next * (a, 6)`
		More than one synonym	`Affects (a, a1)`

AC3 Algorithm

The Arc Consistency Algorithm #3 (AC3) aims to optimise query evaluation by treating it as a constraint satisfaction problem. A more detailed explanation of how AC3 is incorporated into our SPA can be found in Section 3.5.10 ACConstraintGraph and Section 3.5.1.11 ConstraintGraphOptimizer.

Instead of sorting query clauses by some arbitrary metric, it takes a different approach by reducing domains of synonyms involved in the query as much as possible, before performing backtracking to find valid assignments for selected synonyms.

3.5.6.2 Evaluation Criteria

We will use two criteria to decide between using QueryClauseSorter or the AC3 algorithm. The first criteria is the speed of query evaluation with either solution. As speed is one of the key goals in query optimisation, we are likely to use a solution that is able to evaluate complicated queries quickly. While AC3 alone could be considered comparable in speed to the QueryClauseSorter method, with the addition of optimization using ConstraintGraphOptimizer it becomes much faster. This is because the optimization allows us to drop a significant number of rows from the final QueryTable through tree pruning and chain reduction (see Section 3.5.1.11 ConstraintGraphOptimizer for details). By comparison, no matter the manner in which the QueryClauseSorter prioritizes clauses, the final QueryTable will still have more redundant rows than one optimized by AC3 optimizer.

This is balanced by our second criterion which is ease of implementation. This could be how specific our implementation of the solution has to be with regard to the clause in question.

In the case of the QueryClauseSorter, we have to explicitly set the priority of clauses and its synonyms in place for the optimisation to work. Furthermore, for QueryClauseSorter to do so we would have to add additional methods to QueryClause to differentiate clauses, which would break the abstraction and intention of making QueryClause abstract in the first place.

On the other hand, AC3 does not require any knowledge of a clause's type. It only needs the corresponding QueryTable of each clause, which the QueryEvaluator would have had to extract anyway to perform joining if using the QueryClauseSorter method. Moreover, the algorithm itself is well-established and numerous examples of methods to implement and optimize it exist online and in textbooks. This factor alone increases its ease of implementation by a significant amount.

Factoring in the ease of implementation is important because we would want a solution that requires a less specific implementation so we would not have to make many modifications when we have extensions to our SPA scope. For example, introducing a new type of relationship, or a new way of fetching clause data from the PKB could require large overhauls in query optimisation.

3.5.6.3 Evaluation of Options

	Design considerations	
	Speed	**Easy to Implement**
QueryClauseSorter	✓	
AC3	✓	✓

3.5.6.4 Final Decision and Rationale

A key difference between how our SPA uses the QuerySorter and AC3 is that the implementation of the QuerySorter is more granular, meaning we had to differentiate query clauses by their clause type (pattern, with, such that), their relationship type (if relationship can be precomputed or not) as well as the number of synonyms they have. As mentioned in Section 3.5.6.2 Evaluation Criteria, such a detailed implementation would require significant modifications when changes were made to other components.

We initially used the QuerySorter as the implementation was touched on during the lecture on optimisation, but switched to the AC3 algorithm midway through Iteration 3 as we saw how the ease of implementation was better than what QuerySorter had to offer, and better suited our needs as we were introducing a new relationship type for the project extension.

4 Component Interactions

This section describes the interactions between SPA components.

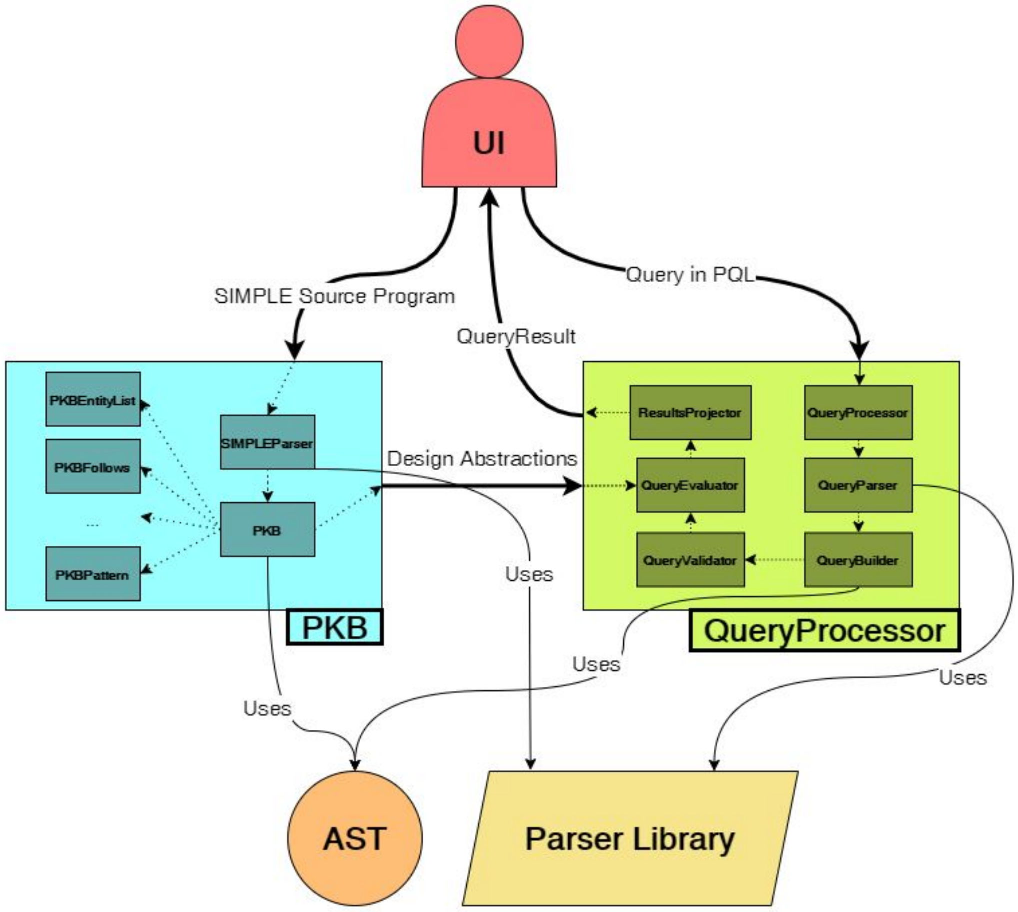


Figure 4.1: Flow of Data Through the Two Main Components

In the early stages of the project, we used rough versions of the above diagram to discuss how the architecture of our SPA would look like. It mainly aided in project planning and communication. In our rough version, we still used colour to demarcate the different components, but we drew the diagrams by hand. Other diagrams were also drawn and we looked at this diagram to help figure out what arrangement/choice of classes would lead to the best overall cohesion and coupling. The UML diagram in Figure 4.1 above helped us communicate the key concepts needed to design the overall architecture.

4.1 QueryProcessor and PKB

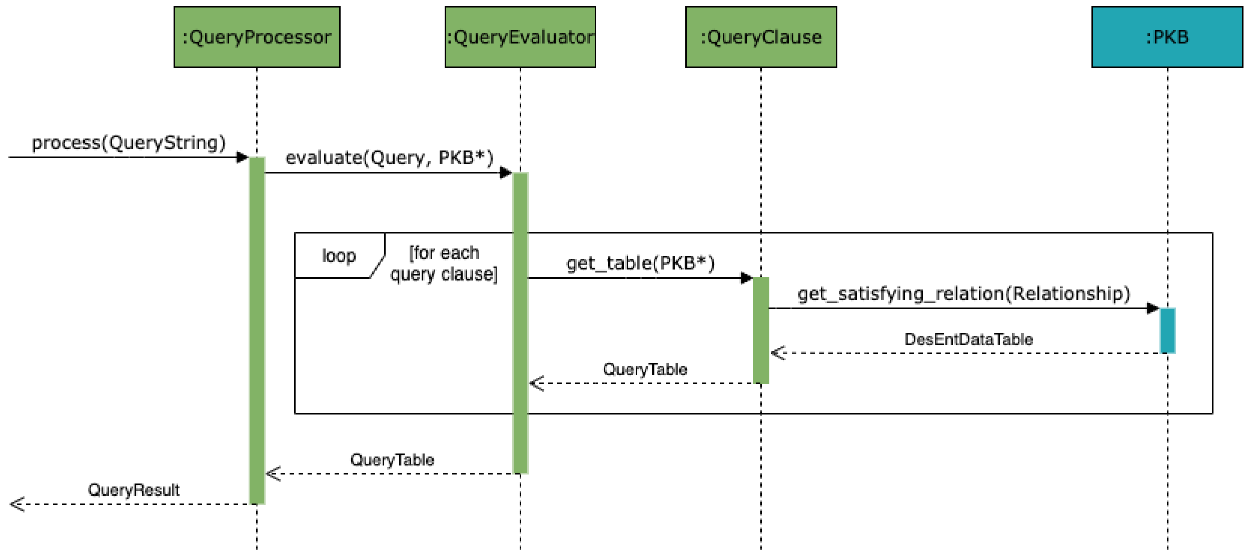


Figure 4.1.1. Interaction Between QueryProcessor and PKB For a Such-That Query Clause

As noted in Section 3.5.5 Design Decision: QueryEvaluator, the only instance of the Query Processor interacting with the PKB is from within the QueryClause class. This change was planned after the submission of Iteration 1 and the above sequence diagram was very helpful during that planning process. We used it to pinpoint exactly which component and which moment in the query evaluation process the Query Processor should interact with the PKB. By doing so, we could also confirm what methods the PKB had to expose as an API for the Query Processor.

The sequence diagram in Figure 4.1.1 above thus served as both a planning tool and a communication tool, since everyone on the team could refer to it to see how the QueryProcessor-PKB interaction worked.

4.2 Subcomponents of QueryProcessor

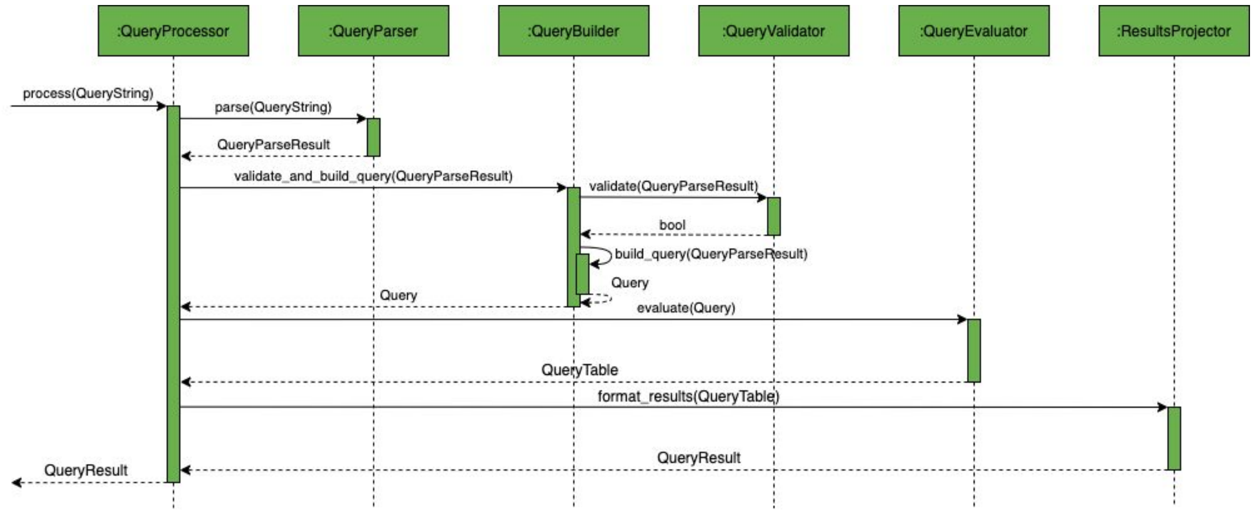


Figure 4.2.1: Interactions Between Subcomponents of the QueryProcessor for Parsing and Evaluating Queries

The sequence diagram in Figure 4.2.1 above shows interactions between subcomponents of the QueryProcessor. This diagram included more specific function calls and return values which allowed us to better plan our API design.

By clearly defining the roles of each Query Processor subcomponent, we also better adhere to the Separation of Concerns and Single Responsibility Principle. This helped us to divide the work better as well; since each component could rely on the abstract APIs of the other components being available, each component could be worked on simultaneously and in isolation by different team members. This helped to accelerate the workflow, allowing us to complete the Query Processor and begin testing earlier.

5 Extension: Branch into Procedures (Bip)

5.1 Overview

Our team implemented inter-procedural NextBip, NextBip*, AffectsBip and AffectsBip* for the general case where a procedure can be called multiple times by other procedures. The first 3 Bip relations adhere to the definitions from the CS3203 wiki page. However, we implemented AffectsBip* as a simple transitive closure of AffectsBip instead due to a lack of time and also in order to focus more effort towards testing the overall system for the full SPA requirements.

Since the Bip definitions are already defined and provided to us on the CS3203 wiki page, we will just provide a brief summary and focus on the details of our implementation.

5.1.1 NextBip, NextBip*

Queries regarding NextBip are handled by the class PKBNextBip, which takes in PKBEntittList and PKBNext, and would use them as blackboxes for computation.

Our definition of NextBip is slightly different from the CS3203 wiki page. For a particular NextBip(a, b) relation, we define the CFGBip to be the control graph **assuming the program starts executing at statement a**. Specifically, NextBip(a, b) is false if statement a is connected to statement b via a BranchOut edge, because if the program starts executing at statement a, then there is no other procedure it can return to via a BranchOut edge. Other than this, the definition mostly follows from the CS3203 wiki, where NextBip(a, b) is defined True for any two nodes a, b that are connected by a direct edge in the CFGBip and similarly, NextBip*(a, b) is defined True if there is a directed path from a to b, **assuming the program starts executing at statement a**.

To handle NextBip, we can simply check which statements can be visited in the next step. The only difference between NextBip(s1, s2) and Next(s1, s2) would be if s1 is a call statement. Hence, we can simply check if s1 is a call statement, and if it is, the only value of s2 that would return true would be the first statement in the procedure called by the call statement.

Then, to precompute all NextBip* relations, we will create the CFGBip graph by taking the CFG (from the Next relation), and adding extra edges from the Call statements to the first statement of the procedure called.

For example, the following SIMPLE program will produce the following CFGBip graph as seen in Figure 5.1.1.1 below:

```
procedure Bill {
1.     x = 1;
2.     call Mary;
3.     x = 2;
4.     call John;
5.     x = 3;
}

procedure Mary {
6.     x = 4;
7.     call John;
8.     x = 5;
}

procedure John {
9.     x = 6;
}
```

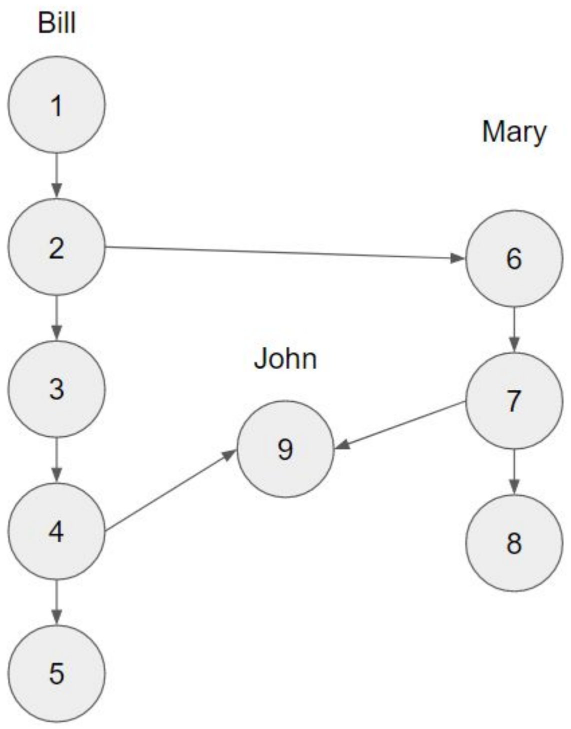


Figure 5.1.1.1: CFGBip Graph from SIMPLE Program

Notice that we have only included the BranchIn edges and not the BranchBack edges. This is because we have included the edges from call statements to the "next" node in the same procedure, so no BranchBack edges need to be added. For example, instead of having a BranchBack edge from Mary's statement 8 to Bill's statement 3, we have just added an edge from statement 2 to 3 to "fast-forward" the function.

Then, we find all transitive relations on the graph using ConnectivityChecker (Section 3.4.1.2), and any NextBip* relation can be queried by checking the transitive closure in ConnectivityChecker.

5.1.2 AffectsBip, AffectsBip*

Queries regarding AffectsBip are handled by the class PKBAffectsBip, which takes in PKBEntittList, PKBModifies, PKBUses, as well as PKBNext, and would use them as blackboxes for computation.

AffectsBip(a, b) is defined between 2 assign statements a, b not necessarily from the same procedure, and there should be a control flow path along the CFGBip such that b uses a variable v that a modifies, and no other assignment statement along the CFGBip path modifies v.

We define AffectsBip*(a, b) to be simply the transitive closure of AffectsBip, which is a deviation from the definition on the CS3203 wiki page.

To compute the AffectsBip, we will construct CFGBip graphs for variables we are interested in.

When an assignment modifies a variable, we want to find a path to another assignment that does not modify that variable. Hence, for that particular variable, we can construct a graph consisting of all valid paths that does not modify that variable.

For example, in the following SIMPLE program, if we would like to check if AffectsBip(1, 5) is true, then, we would need to find a path from statement 1 to 5 such that no statements between modifies the variable x. Hence, this would be the corresponding graph of variable x:

```
1.     x = 1;
2.     if (1 == 1) {
3.            x = 2;
       } else {
4.            y = 1;
       }
5.     target = x;
```

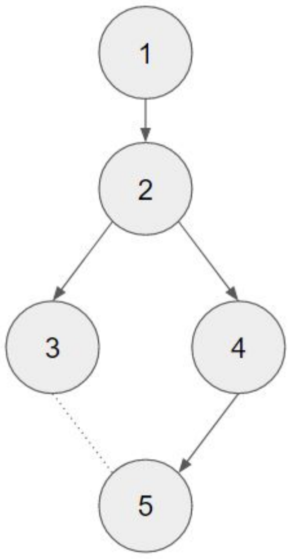


Figure 5.1.2.1: Graph of Variable "x"

Note that there is no edge from 3 to 5 because statement 3 modifies variable x.

To handle BranchIn and BranchOut paths, we can add edges from the call statement to the procedure, and add an edge from the call statement to the next statement in the procedure only if the call statement has the property that it "must modify" the variable. In other words, after the call statement returns, every possible path it can take in the CFGBip contains an assignment or read statement that modifies the variable. Hence, we would like to compute the "must modify" property of every procedure.

A procedure “must modify” a variable if starting from the first statement in the procedure, in all possible paths to the end of the procedure, all paths must contain either an assignment statement or a read statement that modifies that variable, or it must contain another call statement whose procedure “must modify” that variable. This “must modify” property can be computed for all procedures using DFS.

For example, in the following SIMPLE program, this would be the graph generated from variable x:

```
procedure Bill {
1.     x = x;
2.     call Mary;
3.     x = x;
4.     call John;
5.     x = x;
}

procedure Mary {
6.     read x;
}

procedure John {
7.     if (1 == 1) {
8.             x = x;
       } else {
9.             y = y;
       }
}
```

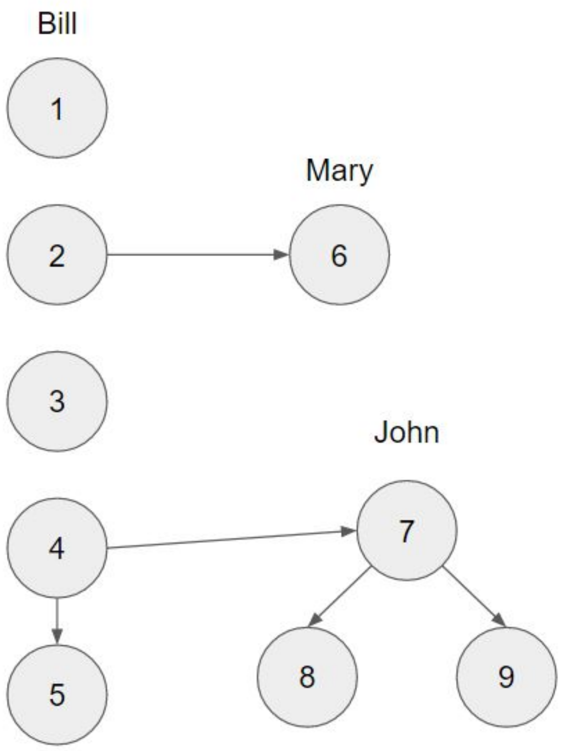


Figure 5.1.2.2: Graph From Variable “x”

In this program, Mary “must modify” x, while John might not, because John could take the path 7 → 9 → end.

There is no edge from 1 → 2 because statement 1 modifies x. There is no edge from 2 → 3 because Mary “must modify” x, hence, the “call Mary” statement is also “must modify” x. Since John might not modify x, we can add an edge from 4 → 5.

Then, to check if AffectsBip(a1, a2) is true, we will find all statements s after a1 in the CFG, and check if there exist a path in the constructed graph from s to a2, and check if Uses(a2, v) is true.

For example, in the program above, to compute AffectsBip(1, 3), we would check if there exists a path from 2 → 3, which there is not, so AffectsBip(1, 3) is false. To compute AffectsBip(3, 8) we would check if there exists a path from 4 → 8, which there is, and since Uses(8, “x”) is true, hence, AffectsBip(2, 8) is true.

Hence, to precompute all AffectsBip relations, we could create up to N different variable graphs for each unique variable modified by an assignment statement, and query the corresponding graph. After computing all AffectsBip relations, the AffectsBip* is simply the transitive closure of that, which is easy to compute using ConnectivityChecker (Section 3.4.1.2).

5.2 Challenges / Affected Components

During our team discussion, we identified that the Bip extensions affects the PKB and QueryProcessor components. In order to adequately tackle the challenge of implementing this extension whilst ensuring that no effort is detracted from the full SPA requirements and testing, we planned the manpower allocation and drew up a timeline for extension development. More details are included in the Extension Schedule Plan in the following section.

During the discussions, the components that we identified that require modification are:
PKB:

- Create classes PKBNextBip and PKBAffectsBip that helps us to retrieve NextBip, NextBip*, AffectsBip and AffectsBip* relations

QueryProcessor:

- QueryParser must accommodate the new type of Bip clauses for form the QueryParseResult object
- QueryValidator has to accommodate semantic error checking (e.g. NextBip*(s, s) is allowed for same synonym s)
- Any optimization algorithms should also include Bip clauses (e.g. group and sort Affects/AffectsBip last)

5.3 Extension Schedule Plan

For this extension, we decided to assign Lim Li and Shawn to the implementation, while Jonathan was responsible for the testing of the extension.

5.3.1 Rationale

We assigned 2 members for implementation and 1 member for the testing of the NextBip, NextBip*, AffectsBip and AffectsBip* Relationships as this arrangement allows us to have the extensions group to work on the extension while the remaining members can focus on meeting the advanced SPA requirements. We also wanted to increase communication and collaboration among the extension team so we intentionally kept the team size small.

Another aspect of this decision is based on the strengths of each team member. Lim Li and Shawn were assigned to implement the main logic of the extension due to their strong knowledge of their respective sub-systems (PKB and QueryProcessor) and Jonathan was assigned to implement testing of the extension due to his experience in testing from his role as the Test IC.

5.3.2 Schedule

Our plan was to set aside Iteration 3.1 to ensure the advanced SPA requirements have been fully implemented and tested extensively. Once the full SPA requirements were implemented, we kept a working copy of our program that could answer full SPA requirements while Lim Li, Jonathan and Shawn work on implementing the NextBip, NextBip*, AffectsBip and AffectsBip* Relationships extension. This gave us the choice to revert back to the previous code if we decided to abort the development of this extension. In mini-Iteration 3.2, Lim Li and Shawn worked on the implementation of the extension within their respective sub-systems. Jonathan then created the unit, integration and system test cases simultaneously with the development team. Additionally, the extension team was responsible for the code quality and documentation of the extension.

The following is a detailed plan of the development of the extension.

Iteration	**Tasks**	**Members**
3.1	Achieve full SPA requirements	All
3.2	Implement NextBip / NextBip* relationship	Lim Li, Shawn
3.2	Implement AffectsBip / AffectsBip* relationship	Lim Li, Shawn
3.2	Unit and Integration Testing	Jonathan
3.2	System testing to ensure results are correct	Jonathan
3.2	Code quality and documentation	Lim Li, Jonathan, Shawn

5.4 Test Plan

Iteration			Iteration 3		
Mini-Iteration			M1	M2	
Week			10	11	12
Unit Tests	Iteration 3	- PKBNextBib	🟩		
		- PKBAffectsBib		🟩	
Integration tests	SIMPLEParser (PKB - Parser Library)				🟩
	QueryParser (Query Processor - Parser Library)				🟩

5.4.1 Rationale

Similar to our testing plan for our SPA development, we conducted unit testing of newly added extension features when the implementation was done. Unit testing was conducted on the PKB and QueryProcessor, which were the sub-components that were affected by the addition of the Bip relationship. After this, integration testing between SPA components, between PKB components and between QueryProcessor components was conducted to ensure that the changes in the different components worked well.

Regression testing was conducted after to ensure that the system could still fulfil the full SPA requirements.

After which, system testing was performed by designing sample SIMPLE programs and queries with extension features to test the system's behaviour.

5.4.2 System Testing

For extensions, we have created 1 sample SIMPLE program and 2 sample test query files for system testing.

Moreover, since AffectsBip and AffectsBip* have similar behavior to Affects for source programs without call or read statements, we did some cursory checks to ensure this by replacing "Affects" to "AffectsBip" and "Affects*" to "AffectsBip*" for system tests that had no call or read statements, and then checking that the query results were the same.

NextBip* is also a superset of Next*, so we could check if the pairs of NextBip* is a superset of the Next* pairs. Moreover, if we restrict the NextBip* to statements in the same procedure, the list of pairs should be exactly identical to Next*.

The sample SIMPLE program and testing features are listed below.

```
procedure proc1 {
    if (1 == 1) then {
        call proc2;
    } else {
        call proc3;
    }
    x = 1;
    call proc2;
}

procedure proc2 {
    while (1 == 1) {
        call proc3;
    }
    x = 1;
    call proc3;
}

procedure proc3 {
    x = 1;
}
```

	Test Purpose	**Required Test Inputs**	**Expected Test Results**
Test case 1	Check for call stmt to next proc	Select BOOLEAN such that NextBip(2,6)	TRUE
Test case 2	Check call stmt in same proc not valid	Select BOOLEAN such that NextBip(2,4)	FALSE
Test case 3	Check NextBip* in different procedure	Select BOOLEAN such that NextBip*(1,10)	TRUE
Test case 4	Check while loop	Select BOOLEAN such that NextBip*(7,7)	TRUE

5.5 Conclusion

In conclusion, we believe that we sufficiently studied the extension requirements, planned appropriate schedules, and conducted extensive testing to ensure that we pushed a robust and well-tested extension alongside the other demanding requirements of Iteration 3.

6 Documentation and Coding Standards

Our naming convention for variables and functions in our documentation uses snake_case. For abstract data types and classes, we use pascalCase. The data types we use in our documentation are the same as those given in the grammar rules of the SPA requirements (e.g. stmtRef, synonym etc.).

Our team adopted the Google C++ style guide for our codebase.

To enhance correspondence between abstract APIs and their concrete API counterparts, we have used the same abstract data types in our documentation and header files. For documentation, our team set up Doxygen to generate documentation for our code base that could be referenced externally without looking at the code files directly.

The following section describes the documentation and coding standards we have adopted for our project.

6.1 Documentation Standards

6.1.1 General

- Every section follows the same numbering scheme
- Every diagram is labelled with a figure number for the reader's ease of reference

6.1.2 Structure

In general, major sections follow the following structure (headings may vary in exact wording by the same idea is used):

- Overview
- Rationale
- Implementation details
- Design considerations

6.1.3 Diagram and Tables Design

Diagram Design
We use the same colour to represent a component in all diagrams throughout the report.

Table Design
Tables follow a standard format. Headers are a darker grey colour while the table contents alternate between 2 similar shades of grey for easier reading.

Title		
Sample Text		

6.1.4 Documentation Standards for Abstract APIs

The diagram in Figure 6.1.4.1 below illustrates the documentation standard for our Abstract API. The arrows indicate what the component in the Abstract API corresponds to.

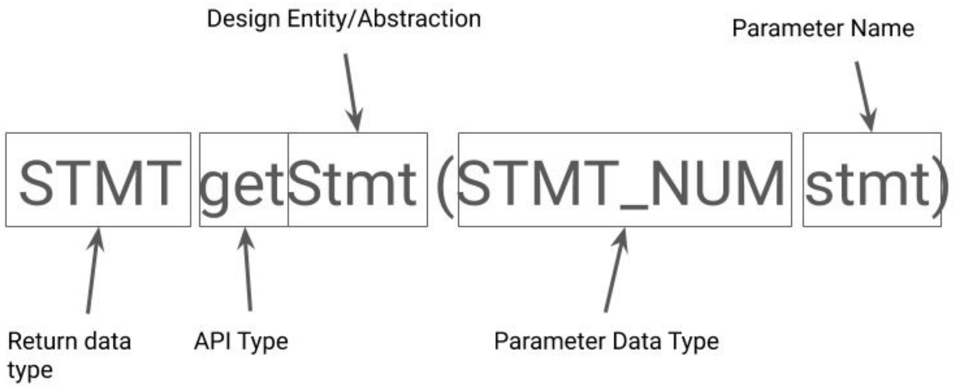


Figure 6.1.4.1: Documentation Explanation for Abstract API

The table below gives a more detailed explanation on what each component of our Abstract API describes.

Abstract APIs Components	**Format (Examples)**	**Explanation**
Return Data Type	snake_case in UPPER_CASE (e.g. BOOLEAN, VOID, STMT_NUM)	Describes the return type in a generic way such that different data structures could be used in the actual implementation.
API Type	camelCase (eg. add, check)	Describes the action of the API to the PKB. camelCase is chosen as descriptions might run for many words.
Design Entity/Abstraction	PascalCase (e.g. Stmt, Follows, Parent)	Describes the design entity or design abstraction.
Parameter Data Type	snake_case in UPPER_CASE (e.g. STMT_NUM)	Describes the parameter type in a generic way such that different data structures could be used in the actual implementation.
Parameter Name	camelCase (e.g. leftStmt)	Describes the semantic value of the item expected by the PKB API.

6.1.5 Mapping Abstract API to Concrete API

When mapping the abstract API data types to the concrete C++ data types we would be using, we chose to make the abstract data types as human-readable as possible. Furthermore, we chose to map multiple C++ concrete data types (e.g. `vector<int>`, `unordered_set<int>`) to a single abstract data type (e.g. LIST_OF_STMT_NUM). Both of these decisions helped us to design the overall program architecture, sequences of calls, and relationships between sub-components without being bogged down by the restrictions of the concrete C++ data types and their (non-)existing API.

When it came to designing our own concrete C++ classes, we chose to follow the naming of already existing abstract API data types. For example, the abstract API data type PKB_FOLLOWS would translate to the `PKBFollows` class in our code. This similarity helped us transition from the design and planning stage to the implementation stage with greater ease and improved our efficiency.

The following table shows a mapping of how we mapped our abstract API data types (present in Section 9 Documentation of Abstract APIs) to concrete C++ data types. Abstract API data types that were translated to custom C++ classes (e.g. PKB_FOLLOWS -> `PKBFollows`) are not included in the table for brevity's sake.

Abstract Type	Concrete Types
STMT_NUM	`int`
VAR_NAME	`string`
PROC_NAME	`string`
LIST_OF_STMT_NUM	`vector<int>, unordered_set<int>`
LIST_OF_VAR_NAME	`vector<string>, unordered_set<string>`
LIST_OF_PROC_NAME	`vector<string>, unordered_set<string>`
DES_ENT	Enumeration `{ STMT, READ, PRINT, CALL, WHILE , IF, ASSIGN, VARIABLE, CONSTANT, PROCEDURE, PROGLINE }`
DES_ENT_DATA	`string`
DES_ENT_DATA_DOMAIN	`vector<string>`
DES_ENT_DATA_TABLE	`vector<vector<string>>`
PATTERN	Enumeration `{ ASSIGN, WHILE, IF }`
BOOLEAN	`bool`
VOID	`void`

6.2 Coding Standards

Our team has adopted the Google C++ coding standard for our code base. We enforced the style rules using clang-format and clang-tidy to ensure that the codebase is well-formatted throughout the software development process.

You may refer to the Google C++ Style Guide for more details.

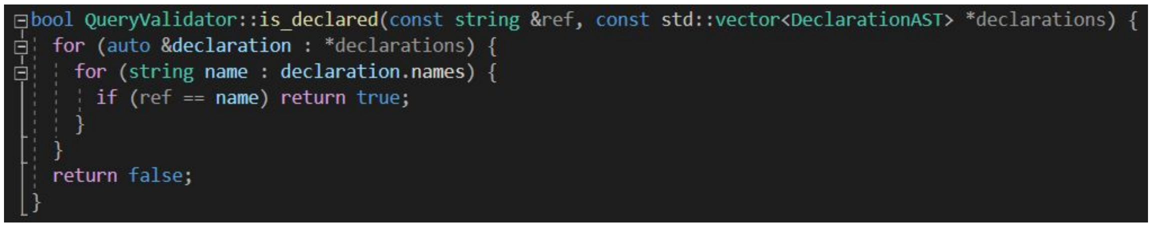
6.2.1 General

- At the end of every file, there will be a new line
- Each line should have a maximum of 120 characters

6.2.2 Curly Braces Indentation

- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line
- The close curly brace is either on the last line by itself or on the same line as the open curly brace (for empty statement lists)
- There should be a space between the close parenthesis and the open curly brace

Example:



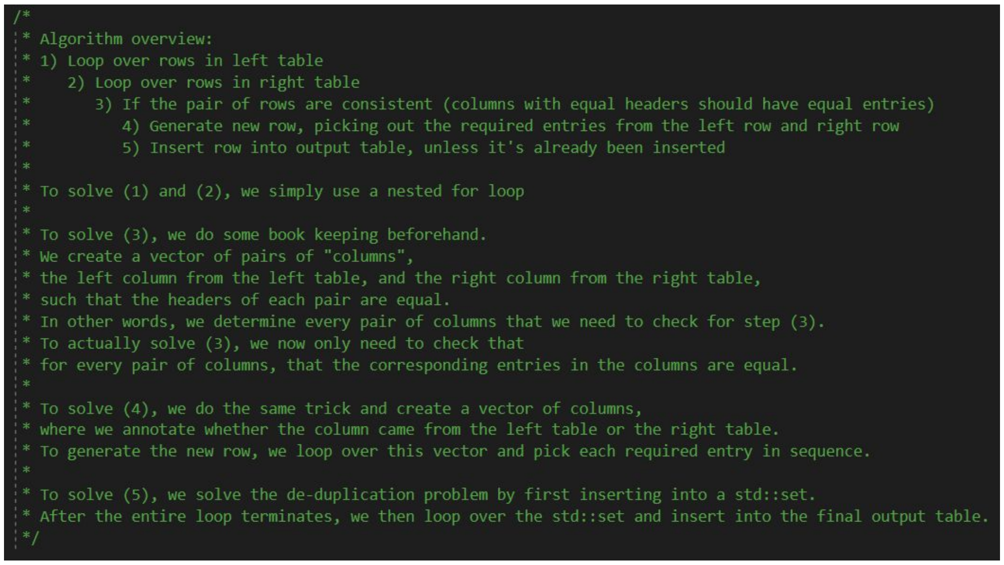
```
bool QueryValidator::is_declared(const string &ref, const std::vector<DeclarationAST> *declarations) {
    for (auto &declaration : *declarations) {
        for (string name : declaration.names) {
            if (ref == name) return true;
        }
    }
    return false;
}
```

6.2.3 Naming Conventions

Type	Naming Convention (example)
Variables	Must be in snake_case (e.g. `such_that_clauses`)
Functions	Must be a verb followed by nouns in camelCase (e.g. `retrieveAllStmts()`)
Class Names	Must be nouns and written in PascalCase (e.g. `QueryValidator`)
Enumerations, Constants	Must be all uppercase (e.g. `SYNONYM`)

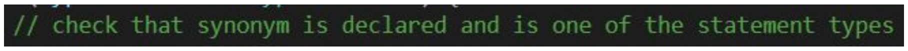
6.2.4 Comments

- All multiple comments are required to use slash followed by an asterisk and close with an asterisk followed by a slash (/* [Comments] */)
  Example:



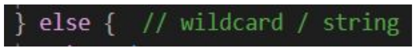
```
/*
 * Algorithm overview:
 * 1) Loop over rows in left table
 *    2) Loop over rows in right table
 *       3) If the pair of rows are consistent (columns with equal headers should have equal entries)
 *          4) Generate new row, picking out the required entries from the left row and right row
 *          5) Insert row into output table, unless it's already been inserted
 *
 * To solve (1) and (2), we simply use a nested for loop
 *
 * To solve (3), we do some book keeping beforehand.
 * We create a vector of pairs of "columns",
 * the left column from the left table, and the right column from the right table,
 * such that the headers of each pair are equal.
 * In other words, we determine every pair of columns that we need to check for step (3).
 * To actually solve (3), we now only need to check that
 * for every pair of columns, that the corresponding entries in the columns are equal.
 *
 * To solve (4), we do the same trick and create a vector of columns,
 * where we annotate whether the column came from the left table or the right table.
 * To generate the new row, we loop over this vector and pick each required entry in sequence.
 *
 * To solve (5), we solve the de-duplication problem by first inserting into a std::set.
 * After the entire loop terminates, we then loop over the std::set and insert into the final output table.
 */
```

- All single comments are required to use double slashes (// [Comments])
  Example:



```
// check that synonym is declared and is one of the statement types
```

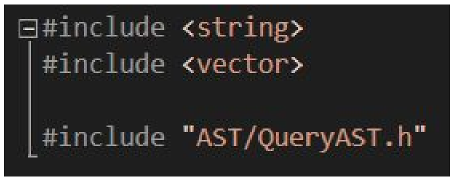
- Any comments at the end of line should have 2 spaces
  Example:



```
} else {  // wildcard / string
```

6.2.5 Include Libraries

- There should only be 1 include in all cpp files and that should be the associating header file of the class
- There should be 2 groups for includes in header files: 1 group for C++ libraries, 1 group for user-defined libraries
- These 2 groups should be in alphabetical order and separated with 1 line
  Example:

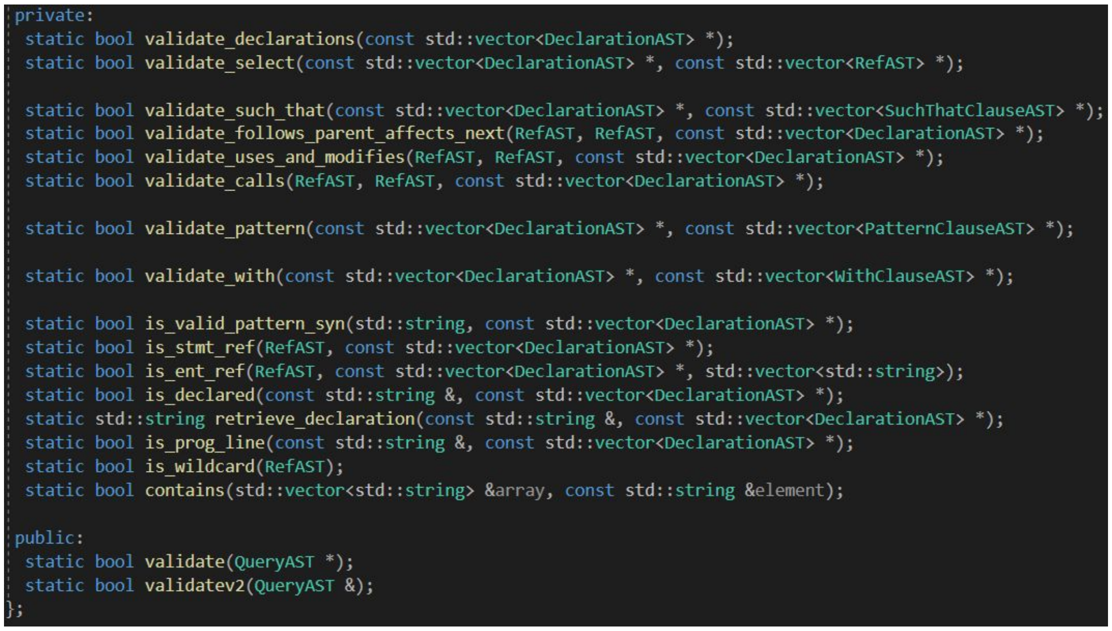


```
#include <string>
#include <vector>

#include "AST/QueryAST.h"
```

6.2.6 Ordering of methods in header and cpp files

- In all cpp files, constructors take precedence over private methods in alphabetical order followed by public methods in alphabetical order and then any operator overloading methods
- In all header files, private methods take precedence over private methods
  Example:



```
private:
  static bool validate_declarations(const std::vector<DeclarationAST> *);
  static bool validate_select(const std::vector<DeclarationAST> *, const std::vector<RefAST> *);

  static bool validate_such_that(const std::vector<DeclarationAST> *, const std::vector<SuchThatClauseAST> *);
  static bool validate_follows_parent_affects_next(RefAST, RefAST, const std::vector<DeclarationAST> *);
  static bool validate_uses_and_modifies(RefAST, RefAST, const std::vector<DeclarationAST> *);
  static bool validate_calls(RefAST, RefAST, const std::vector<DeclarationAST> *);

  static bool validate_pattern(const std::vector<DeclarationAST> *, const std::vector<PatternClauseAST> *);

  static bool validate_with(const std::vector<DeclarationAST> *, const std::vector<WithClauseAST> *);

  static bool is_valid_pattern_syn(std::string, const std::vector<DeclarationAST> *);
  static bool is_stmt_ref(RefAST, const std::vector<DeclarationAST> *);
  static bool is_ent_ref(RefAST, const std::vector<DeclarationAST> *, std::vector<std::string>);
  static bool is_declared(const std::string &, const std::vector<DeclarationAST> *);
  static std::string retrieve_declaration(const std::string &, const std::vector<DeclarationAST> *);
  static bool is_prog_line(const std::string &, const std::vector<DeclarationAST> *);
  static bool is_wildcard(RefAST);
  static bool contains(std::vector<std::string> &array, const std::string &element);

public:
  static bool validate(QueryAST *);
  static bool validatev2(QueryAST &);
};
```

7 Testing

Before describing our tests and test plans, we describe our testing workflow. For unit testing and integration testing, we used the provided CMake targets to run all tests. For system testing, we utilized the autotester executable provided. To encourage the organisation of system tests, query test files were written in this form:

```
<target source program file>.<description of queries>.queries.txt
```

Example:

```
calculatepi.follows.queries.txt
calculatepi.parent.queries.txt
```

We wrote two scripts, one to compile query .txt files with the same target source program into a single query file, and another (shown in the snippet below) to run the autotester on multiple sources and compiled query files.

```python
def main():
   arguments = docopt(__doc__)
   print('Arguments:')
   print(arguments)
   print()

   print('Finding autotester...')
   autotester_candidates = find_autotester()
   autotester = autotester_candidates[0]
   print('Using autotester:', autotester[1])
   print()

   print('Rebuilding autotester from build dir:', autotester[0])
   rebuild(autotester[0])
   print()
```

7.1 Test Plan

System test plans are more involved and detailed in Section 7.4.

Unit tests were written every time we introduced a new component. It typically occurred early in each mini-iteration after implementing new components required for the scope of said mini-iteration.

Integration tests were only done in Iteration 1. More details in Section 7.3.

Iteration			Iteration 1					Iteration 2			Iteration 3		
Mini-Iteration			M1		M2	M3		M1	M2		M1	M2	
Week			3	4	5	6	R	7	8	9	10	11	12
Unit Tests	Iteration 1	- PKBEntityList - PKBFollows - PKBParent	🟥										
		- PKBUses - PKBModifies			🟥								
		- PKBPattern (assign)				🟥							
		- QueryValidator	🟪		🟪								
		- QueryBuilder			🟪								
		- QueryEvaluator				🟪	🟪						
	Iteration 2	- PKBNext							🟦				
		- PKBCalls - PKBPattern (while, if)								🟦			
		- QueryValidator							🟪				
		- QueryBuilder							🟪				
		- QueryEvaluator								🟪			
	Iteration 3	- PKBAffects									🟩		
		- PKBNextBip										🟩	
		- PKBAffectsBip											🟩
Integration tests	SIMPLEParser (PKB - Parser Library)				🟧		🟧						
	QueryParser (Query Processor - Parser Library)				🟧		🟧						

7.2 Unit Testing

For each pull request (PR), the author of the PR would include the unit tests for new (sub-)components in the PR. The unit test cases extensively cover the functionalities of each sub-component. The reviewer would review both the logic and the test cases in the PR. Finally, the QA lead would give a final look at the unit test cases before merging. All unit test cases are automatically run before merging.

If any bugs are found after the PR is merged, then the PR that fixes the bug would include its regression test in the unit test cases.

There is no plan for unit testing; we simply set out a group requirement that all components are unit tested extensively before we integrate it.

7.2.1 PKB Examples

	Test Purpose	**Required Test Inputs**	**Expected Test Results**

PKBUses	We mock the SimpleAST for test cases to ensure that we only test the correctness of the PKBUses class.
	
	Verify that the PKBUses class can correctly evaluate if some statement or procedure uses some variable.
		Mock SimpleAST: To feed into the constructor of PKBPattern, and to test against. E.g.:
		`5. ...`
		`6. while(x < y) {`
		`7.     print(bbb); }`
		`8. ...`
		
		A statement number/procedure name and a variable name. E.g:
		- Statement 7
		- Variable “bbb”
			Returns true since Uses(7, bbb) is true.

PKBPattern	We mock the SimpleAST for test cases to ensure that we only test the correctness of the PKBPattern class.
	
	Verify that the PKBPattern class can correctly match specific assignment statements against specific assignment patterns.
		Mock SimpleAST: To feed into the constructor of PKBPattern, and to test against. E.g.:
		` 9. ...`
		`10. ff = (69420 + g) % (9 - ff)`
		`11. ...`
		
		Mock SPAPatternA: A class containing information about the assignment pattern to be matched. This includes LHS variable, RHS expression, and whether we match sub-expressions. E.g.:
		`(_, _”9 - ff”_)`
			Returns true when testing statement 10 against the example pattern.

7.2.2 Query Processor Examples

	Test Purpose	**Required Test Inputs**	**Expected Test Results**
Query Validator	We mock the QueryParseResult for each test case in order to ensure that successive test cases are not affected by the execution of previous test cases. Verify that the case where the second parameter provided in the Modifies relationship is correctly identified as invalid	QueryParseResult containing a Modifies(p, a) where "p" is declared as a synonym for procedure and "a" is declared as a synonym for assignment	Query Validator should return false
Query Evaluator	We mock the PKB for each test case in order to ensure that successive test cases are not affected by the execution of previous test cases. If a single instance of the PKB is used and it is modified by a test case, the results of successive test cases might be affected. We mock the PKB for each relation (eg. PKBFollowsMock, PKBParentMock) to independently verify that each relation is handled appropriately.	Mock PKB containing: ● statements 1 and 2 ● relationship Follows(1, 2) Query: `Select s1 such that Follows(s1, s2)`	QueryResult, a list containing a single string entry "1"

7.3 Integration Testing

Our architecture now only consists of two main components, the PKB and the Query Processor. Therefore, integration tests between these two components are synonymous with system testing and were done there.

The only other "integration" we would have needed to test is between the Parser Library, AST data structure, and the main components' Parser sub-component. This would have been equivalent to unit-testing the sub-components SIMPLEParser and QueryParser, **which was done quite early in development**. We designed and implemented our parser components with respect to the final project requirements. Therefore, no plans were created for integration testing after we had tested the integration extensively once.

7.3.1 SIMPLEParser Examples

	Test Purpose	Required Test Inputs	Expected Test Results

SIMPLEParser	Verify that the parser can successfully parse a program with one print statement.	Syntactically correct SIMPLE program to feed into the parser:
		
		`"procedure test {\n"`
		`"print 5;\n"`
		`"}\n");`
			An equivalent SIMPLE AST. Omitted as it is verbose.

SIMPLEParser	Verify that the parser fails to parse a syntactically incorrect program (procedure name cannot start with number).	Syntactically incorrect SIMPLE program to feed into the parser: `“procedure 1testproc {  }”`	Parse error at the exact location (line 1 character 11), with a report on the expected token: `ExpectedToken::PROCNAME`

7.3.2 QueryParser Examples

	Test Purpose	Required Test Inputs	Expected Test Results
QueryParser	Verify that the parser can successfully parse a query with one declaration (assign) and one select clause.	A syntactically valid query: `"assign a; Select a"`	An equivalent manually constructed QueryParseResult. Omitted as it is verbose.
QueryParser	Verify that the parser fails to parse a syntactically incorrect query (“stmt#” not “stmt”).	A syntactically invalid query: `"assign a; procedure p; Select <a.stmt, ..."`	Parse error at the exact location (line 1 character 37), with a report on the expected token: `ExpectedToken::HASH`

7.4 System Testing

System testing is split into two sections: Correctness and Efficiency. We describe the tests for these two sections, before detailing the plans to implement these tests.

7.4.1 Correctness

7.4.1.1 Queries

To verify the correctness of the SPA, we take a systematic approach in system testing, separating out tests into groups by the number of joins and clause graphs of the query. A table summary with examples is as follows:

Join Type	Clause Graph		Identifier
1. One Clause (No Join)	a. Single clause		1.a
	b. Invalid clauses		1.b
2. Multiple joins (Select all used synonyms)	a. Sharing one common synonym (1 Join)	i. Two clauses chain `Select <a,v> such that Uses(a, v) pattern a(v, _)`	2.a.i
		ii. One reflexive clause `Select <s> Next*(s, s)`	2.a.ii
	b. Sharing two common synonyms (2 Joins)	i. Three clauses chain `Select <a, v1, v2> such that Uses(a, v1) pattern a(v1, _) with v1.varName = v2.varName`	2.b.i
		ii. Two clauses chain, one reflexive `Select <s, w> such that Next*(s, s) and Parent(w, s)`	2.b.ii
	c. Sharing three or more common synonyms	i. 3-Cycle `Select <s1, v1, a1> such that Modifies(s1, v1) pattern a1(v1, _) with a1.stmt# = s1.stmt#`	2.c.i
		ii. Other arbitrary graphs (Multigraphs, 4-Cycle etc.) `Select <a1, c1, p1, v1> such that Next*(a1, c1) with c1.procName = p1.procName such that Uses(p1, v1) pattern a1(v1, _)`	2.c.ii
	d. Partitions	i. Combinations of connected clause components `Select <a,v, s, w> such that Modifies(s1, v1) pattern a1(v1, _) with a1.stmt# = s1.stmt# such that Next*(s, s) and Parent(w, s)`	2.d.i
3. Multiple joins with projection (Select some used synonyms)	a. Sharing one common synonym	i. Two clauses `Select <a> such that Uses(a, v) pattern a(v, _)`	3.a.i
	b. Sharing three or more common synonyms	i. Other arbitrary graphs (Multigraphs, 4-Cycle etc.) `Select BOOLEAN such that Next*(a1, c1) with c1.procName = p1.procName such that Uses(p1, v1) pattern a1(v1, _)`	3.b.i
	c. Partitions	i. Combinations of connected clause components `Select <a, w> such that Uses(a, v) pattern a(v, _) such that Next*(s, s) and Parent(w, s)`	3.c.i

As seen, the three main levels of system tests are also ordered in terms of complexity, and the correctness of each successive level of system tests is dependent on the correctness of the previous level. Each level attempts to verify correctness at three different levels:

1. Correctly retrieve tables from the PKB
2. Join logic is correct
3. Projection logic is correct

Once the first level check is done, we no longer concern ourselves (too much) with the type of clause (suchthat, with, pattern) we use, since the join logic only uses tables to perform the join. Despite that, we still try to make sure our clause-type usage per query is diverse for subsequent tests.

In each level, we introduce finer granularity of system tests by splitting system tests into "Clause Graph", which in turn affects the number and sequence of joins we do. The rationale behind this is symmetric; evaluation of queries with complex clause-graph structures can only be correct if the evaluation of one or two clauses is also correct. Here, "partitions" refer to the groups of clauses that (transitively) share common synonyms, or using the graph analogy, connected components.

As an illustration, take the query example from 2.d.i in the table above. The graph that is generated is shown in Figure 7.4.1.1.1 below.

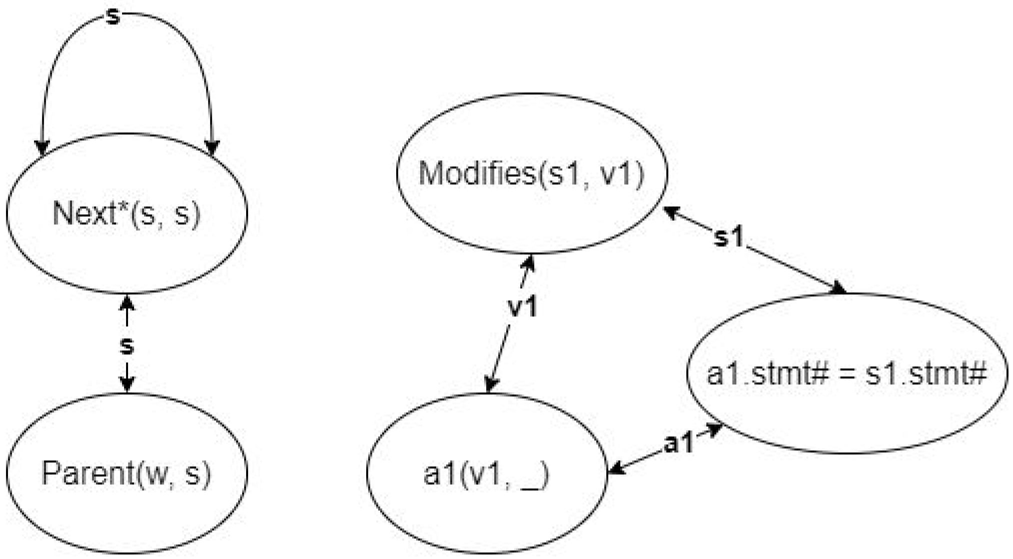


Figure 7.4.1.1.1: Generated graph from query from 2.d.i

Edges are labelled with the synonym that is responsible for the edge. There are two partitions, and note that the two partitions actually come from the query example in 2.b.ii (Chain) and 2.c.i

(3-Cycle). The correctness of this query is thus dependent on the correctness of the evaluation of simpler graph constructs in 2.b.* and 2.c.*.

In the final third level, we take queries in the previous level and remove some or all (in which case we select BOOLEAN) of the selected synonyms. As an example, for the query in 2.a.i:

```
Select <a,v> such that Uses(a, v) pattern a(v, _)
```

A full projection test will mean that we construct a query for each subset of {a, v} in the the powerset of {a, v}:

```
Select BOOLEAN such that Uses(a, v) pattern a(v, _)
Select <a> such that Uses(a, v) pattern a(v, _)
Select <v> such that Uses(a, v) pattern a(v, _)
```

The "Projection" test comes after the pure "Join" test because of two reasons. Firstly, as mentioned, we can test the projection in isolation if we are sure the joins are correct. Secondly and more importantly, is because of how optimization is done internally, where we quickly discard unneeded synonym columns if they no longer affect the joins of future clauses (Section 3.5.1.11 ConstraintGraphOptimizer). We want to make sure that under heavier optimization, we still have correctness.

7.4.4.2 SIMPLE Programs

In this section, we discuss our methodology in creating SIMPLE source programs to test the queries against. All mentioned source programs can be found in Appendix D.

We designed one relatively large SIMPLE source program "calculatepi.source.txt". This source program contains at least one of each design entity inside, with a good many variables and procedures. Most of the queries testing for join and projection use this source program, and have a non-trivial number of rows in most of the tables retrieved for each clause.

In the case where the program is not sufficient to meet the test objective, we create specific SIMPLE programs for the set of queries. This is predominantly for queries testing only a single type of clause (1.*). For instance, in testing for Next(*) or the if/while pattern clauses, the above SIMPLE program has not enough coverage since it only has one of each if and while statement. We then tailor-craft a SIMPLE program to test the correctness of either clause. These are "edgecase_nest.source.txt" and "basic_ifwhilepattern.source.txt".

The "edgecase_nest.source.txt" program consists of every possible combination of if and while statements in a maximum nesting depth of two. This allows us to test the control flow graph generated by PKBNext for a good coverage of possible nesting scenarios. On the other hand, the "basic_ifwhilepattern.source.txt" consists of randomly generated if/while statements at a maximum nesting depth of 3, with randomly generated conditional expression using a set of 10 variables.

Contrived SIMPLE programs are also designed to test for more unusual scenarios, such as with clauses like `v1.varName = p1.procName` or `cnst1.value = prog_line`. These can be found in source program files prefixed with "edgecase", such as "edgecase_with.source.txt".

7.4.2 Efficiency

Firstly, we want to make sure the PKB is when retrieving tables for relations we do not do pre-computation for. This motivates two sets of stress tests, one for Next(*) and one for Affects(*).

For Next(*) stress test, we used a python script to randomly generate a SIMPLE program with these parameters:

- Maximum number of statements per block: 5
- Chance for a statement to be if/while: 0.6
- Chance for a nesting statement to be an if: 0.5
- Maximum nesting level: 5

We run the generator multiple times until we have a SIMPLE program with 500 LOCs. The script also generates the correct output for the simple Next/Next(*) queries with the SIMPLE program:

```
Select <s1, s2> such that Next(s1, s2)
Select <s1, s2> such that Next*(s1, s2)
```

For the Affects(*) stress test, we took a similar approach of randomly generating large SIMPLE programs with many assignments. In fact, we use the exact same source code as with the Next(*) stress test since all statements that are NOT if or while, are dummy assignment statements "`x = x`". However, unlike the Next(*) generator, we do not output the correct query result along with the SIMPLE program because it's difficult. In any case, we only want to measure speed of fetching the table; correctness should have been checked in the previous section.

Finally, we want to design stress tests that have multiple large tables with multiple joins. Note that we **do not concern ourselves with the type of clauses used,** because of how query evaluation is done by using the AC3 algorithm. All clauses are transformed into constraints, afterwhich their origination is no longer important. So the only thing to vary is the size (number of constraints) and order (number of used synonyms) of the query clause-graph, and its structure (number of partitions, cycles and cliques).

For instance, a large query with a tree-like structure might look something like this:

```
Select <s, a3, b7, e1, e6>
             such that Next*(s, a1) and Next*(a1, a2) and … Next*(a9, a10)
             such that Next*(s, b1) and Next*(b1, b2) and … Next*(b9, a10)
             ...
             such that Next*(s, e1) and Next*(e1, e2) and … Next*(e9, e10)
```

The resulting AC3 constraint graph will be a star with 5 long spokes. Under Tree-Pruning optimisation, the query should be evaluated relatively fast with many domains being dropped after pruning. In fact, the final graph should be a star consisting of only the selected synonyms.

On the other hand, an example of a large query with many cycles might look something like this:

```
Select <a1, b1, c1, d1, e1>
             such that Next*(s, a1) and Next*(a1, a2) and … Next*(a9, s)
             such that Next*(s, b1) and Next*(b1, b2) and … Next*(b9, s)
             ...
             such that Next*(s, e1) and Next*(e1, e2) and … Next*(e9, s)
```

The resulting AC3 constraint graph will have a flower shape with 5 petals (cycles) joined at a single vertex, synonym s. Under Chain-Reduction optimisation, the query should be evaluated relatively fast with many constraints being merged after reduction.

After purposefully constructing graphs that we KNOW will be optimised, and verifying that it IS optimized, we construct randomly generated dense graphs of large order (many synonyms). This is our fuzzing tests. This should cover cases like the number of partitions and cliques, just by sheer number of randomly generated queries. Again, correctness is of less importance here.

To perform our fuzzing tests, we randomly generate SIMPLE programs, and randomly generate queries. These are all done using Python scripts. An example is given at the end of Appendix D. Note that the given example is MUCH smaller than what we use in practise, so is only an illustration of how a random program looks like. The queries in the example likewise.

A summary of all the stress tests is the following list:

1) Next(*)
2) Affects(*)
3) Large queries
    a) Large Trees (Tree-Pruning Optimization)
    b) Many Cycles (Chain-Reduction Optimization)
    c) Fuzzing

7.4.3 Plan

Iteration				Iteration 1					Iteration 2			Iteration 3		
Mini-Iteration				**M1**		**M2**	**M3**		**M1**	**M2**		**M1**	**M2**	
Week				**3**	**4**	**5**	**6**	**R**	**7**	**8**	**9**	**10**	**11**	**12**
Correctness System Test Level	**1.***	**1.a**			🟥	🟥	🟥			🟥				🟥
		1.b			🟥	🟥	🟥							
	2.*	**2.a**	**2.a.i**						🟦	🟦				
			2.a.ii						🟦	🟦				
		2.b	**2.b.i**						🟦	🟦				
			2.b.ii							🟦	🟦			
		2.c	**2.c.i**								🟦	🟦		
			2.c.ii									🟦	🟦	
		2.d	**2.d.i**										🟦	🟦
	3.*	**3.a**	**3.a.i**							🟩				
		3.b	**3.b.i**										🟩	
		3.c	**3.c.i**											🟩
Efficiency System Test Program	**1. Next(*)**									🟨	🟨			
	2. Affects(*)													🟨
	3.*	**3.a. Trees**									🟨			
		3.b. Cycles												🟨
		3.c. Fuzzing												🟨

7.4.4 Example

We leave the examples out of the report for brevity, since the SIMPLE programs are pretty long. Refer to Appendix D for these examples of source SIMPLE programs and queries.

8 Discussion

Our team was able to quickly establish our roles and responsibilities at the beginning of the project. We did so by assessing the complexity of algorithms for each component and the programming ability of each individual member. This allowed everyone in the team to be working in a role that is more comfortable for him/her, which in turn results in everyone being able to contribute effectively to the project.

In terms of project management, we were also able to break the project down into weekly iterations, which allowed us to complete the project smoothly ahead of schedule. We also set aside a dedicated weekly discussion timeslot where we came together to discuss solutions at a high level, before breaking off into our smaller teams during the week to do the actual implementation.

A problem that we faced was that there is quite a large discrepancy in programming ability among team members. We overcame this issue by having the stronger team members coming in to help with more complex tasks throughout the development cycle.

Another problem that we faced initially was that we came up with the abstract data types to be used in the program at a rather late stage. As a result, we had to constantly re-work certain components whenever there are small changes to other components. This issue was promptly resolved once we came together to discuss our abstract APIs.

If we were to start the project again, we would discuss our abstract APIs earlier so that we would reduce the number of problems and re-work we had to do whenever changes are made in the implementation.

9 Documentation of Abstract APIs

9.1 AST

- Overview: Stores and exposes a visitor for SIMPLE programs.
- Public Interface:

Operation header	Description
VOID acceptVisitor(VISITOR visitor)	Uses double dispatch to call the appropriate method in the Visitor object for each type of AST node in the class hierarchy.
PROGRAM createProgram(LIST_OF_PROCEDURE procedures)	Creates a SIMPLE program given a list of its procedures.
PROCEDURE createProcedure(LIST_OF_STMT stmtList))	Creates a SIMPLE procedure given a list of statements.
... (and so on, following the ASG)	

9.2 Parser

- Overview: Parser combinators for constructing parsers.
- Public Interface:

Operation header	Description
PARSER makeFun(FUNCTION fun)	Constructs a parser out of the given parsing function.
PARSER makeNonOwning(PARSER_PTR parserPtr)	Constructs a parser out of a pointer of a parser, only dereferencing the pointer as late as possible.
PARSER makeLift(PARSER parser, FUNCTION fun)	Constructs a parser that runs the given parser, and transforms the output using the given function.
PARSER makeSeq(PARSER parser, PARSER parser, ...)	Constructs a parser out of the given list of parsers, running each parser in sequence. If any parser in the sequence failed, backtrack to the point before the first parser ran.
PARSER makeOperatorAnd(PARSER parser, PARSER parser)	Like makeSeq, constructs a parser that runs two parsers in sequence. If either parser failed, backtrack to the point before the first parser ran.
PARSER makeOperatorLeft(PARSER parser, PARSER parser)	Like makeOperatorAnd, but discards the right result.
PARSER makeOperatorRight(PARSER parser, PARSER parser)	Like makeOperatorAnd, but discards the left result.
PASER makeAlt(PARSER parser, PARSER parser, ...)	Constructs a parser out of the given list of parsers, trying each parser until a parser succeeds. Once a parser succeeds, return its result. If all parsers fail, fail as well.
PARSER makeTwo(PARSER parser, PARSER parser)	Like makeAlt, but with only 2 arguments.
PARSER makeMany(PARSER parser)	Constructs a parser out of the given parser, that runs the given parser as many times as possible until it fails. Always succeeds as it is ok for the given parser to fail on the first attempt.
PARSER makeSome(PARSER parser)	Constructs a parser out of the given parser, that runs the given parser as many times as possible until it fails. Only succeed if the given parser succeeded at least once.

9.3 PKB

- Overview: extract design entities and abstractions from the SPA program AST; answer queries for these design entities and abstractions
- Public Interface:

Operation header	**Description**
DES_ENT_DATA_DOMAIN retrieveAllDesEnt(DES_ENT desEnt)	Returns a container containing all entities of type DES_ENT.
BOOLEAN testRelation(RELATIONSHIP rs, DES_ENT_DATA ded1, DES_ENT_DATA ded2)	Returns true if (ded1, ded2) satisfy the relationship specified by rs, false otherwise and on invalid inputs.
DES_ENT_DATA_TABLE getSatisfyingRelation(RELATIONSHIP rs)	Returns a table of all DES_ENT_DATA pairs satisfying the relationship specified by rs.
DES_ENT_DATA_TABLE filterSatisfyingRelation(RELATIONSHIP rs, DES_ENT_DATA_DOMAIN dedd1, DES_ENT_DATA_DOMAIN dedd2)	Returns a table of all DES_ENT_DATA pairs* satisfying the relationship specified by rs. *: Pairs are taken from the cartesian product of dedd1 and dedd2.
DES_ENT_DATA_TABLE getAllSatisfyingPattern(PATTERN pat, SPA_PATTERN spat)	Returns a table of all DES_ENT_DATA pairs# satisfying the pattern specified by spat of pattern type pat. #: The tuples may vary depending on the pattern type pat and are as follows: - Assignment: (STMT_NUM, VAR_NAME) - If: (STMT_NUM, VAR_NAME) - While: (STMT_NUM, VAR_NAME)
DES_ENT_DATA_TABLE filterLeftSatisfyingPattern(PATTERN pattern, SPA_PATTERN spaPattern, DES_ENT_DATA_DOMAIN dedd)	Returns a table of all DES_ENT_DATA pairs#* satisfying the relationship specified by rs. #: Refer to `getAllSatisfyingPattern`. *: First element of pairs must come from dedd
DES_ENT_DATA_TABLE filterSatisfyingPattern(PATTERN pattern, SPA_PATTERN spaPattern, DES_ENT_DATA_DOMAIN dedd1, DES_ENT_DATA_DOMAIN dedd2)	Returns a table of all DES_ENT_DATA pairs#* satisfying the relationship specified by rs. #: Refer to `getAllSatisfyingPattern`. *: Pairs are taken from the cartesian product of dedd1 and dedd2.

9.4 PKBEntityList

- Overview: Extracts data entities from the AST and answer queries regarding them
- Public Interface:

Operation header	**Description**
LIST_OF_STMT_NUM retrieveAllStmts()	Returns a list of all statements of the program.
LIST_OF_STMT_NUM retrieveAllCalls()	Returns a list of all call statements of the program.
LIST_OF_STMT_NUM retrieveAllAssignments()	Returns a list of all assignment statements of the program.
LIST_OF_STMT_NUM retrieveAllIfs()	Returns a list of all if statements of the program.
LIST_OF_STMT_NUM retrieveAllWhiles()	Returns a list of all while statements of the program.
LIST_OF_STMT_NUM retrieveAllReads()	Returns a list of all read statements of the program.
LIST_OF_STMT_NUM retrieveAllPrints()	Returns a list of all print statements of the program.
LIST_OF_PROC_NAME retrieveAllProcedures()	Returns a list of all procedure names of the program.
LIST_OF_VAR_NAME retrieveAllVariables()	Returns a list of all variable names of the program.
PROCEDURE getProc(PROC_NAME procName)	Returns the AST node of the procedure with name p. Returns NULL if no procedure has name equals p.
STMT getStmt(STMT_NUM stmtNum)	Returns the AST node of the statement with statement number s. Returns NULL if no statement has statement number equals s.

9.5 PKBFollows

- Overview: Extracts Follows(*) relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_FOLLOWS createPKBFollows(AST ast)	Creates a PKB_FOLLOWS.
BOOLEAN isFollows(STMT_NUM s1, STMT_NUM s2)	Returns true if Follows(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isFollowsS(STMT_NUM s1, STMT_NUM s2)	Returns true if Follows*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.

9.6 PKBParent

- Overview: Extracts Parent(*) relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_PARENT createPKBParent(AST ast)	Creates a PKB_PARENT.
BOOLEAN isParent(STMT_NUM s1, STMT_NUM s2)	Returns true if Parent(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isParentS(STMT_NUM s1, STMT_NUM s2)	Returns true if Parent*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.

9.7 PKBUses

- Overview: Extracts Uses relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_USES createPKBUses(AST ast, PKB_ENTITY_LIST pkbEntityList)	Creates a PKB_USES.
BOOLEAN isUses(STMT_NUM s, VAR_NAME v)	Returns true if Uses(s, v), false otherwise. Returns false if s is an invalid statement number or v is an invalid variable name.
BOOLEAN isUses(PROC_NAME p, VAR_NAME v)	Returns true if Uses(p, v), false otherwise. Returns false if p is an invalid procedure name or v is an invalid variable name.

9.8 PKBModifies

- Overview: Extracts Modifies relations from the AST and answer queries regarding them
- Public Interface:

Operation header	**Description**
PKB_MODIFIES createPKBModifies(AST ast, PKB_ENTITY_LIST pkbEntityList)	Creates a PKB_MODIFIES.
BOOLEAN isModifies(STMT_NUM s , VAR_NAME v)	Returns true if Modifies(s, v), false otherwise. Returns false if s is an invalid statement number or v is an invalid variable name.
BOOLEAN isModifies(PROC_NAME p, VAR_NAME v)	Returns true if Modifies(p, v), false otherwise. Returns false if p is an invalid procedure name or v is an invalid variable name.

9.9 PKBPattern

- Overview: Pre-computes information needed to answer queries regarding patterns (Assignment, If, While).
- Public Interface:

Operation header	**Description**
PKB_PATTERN createPKBPattern(AST ast, PKB_ENTITY_LIST pkbEntityList)	Creates a PKB_PATTERN.
BOOLEAN isAssignPatternMatch(STMT_NUM s, SPA_PATTERN spat)	Returns true if s is an assignment statement and matches the pattern specified in spat, false otherwise. Returns false if s is not an assignment statement or an invalid statement number.
BOOLEAN isIfPatternMatch(STMT_NUM s, SPA_PATTERN spat)	Returns true if s is an if statement and matches the pattern specified in spat, false otherwise. Returns false if s is not an if statement or an invalid statement number.
BOOLEAN isWhilePatternMatch(STMT_NUM s, SPA_PATTERN spat)	Returns true if s is an while statement and matches the pattern specified in spat, false otherwise. Returns false if s is not an while statement or an invalid statement number.

9.10 PKBCalls

- Overview: Extracts Calls relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_CALLS createPKBCalls(AST ast, PKB_ENTITY_LIST pkbEntityList)	Creates a PKB_CALLS.
BOOLEAN isCalls(PROC_NAME p1, PROC_NAME p2)	Returns true if Calls(p1, p2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isCallsS(PROC_NAME p1, PROC_NAME p2)	Returns true if Calls*(p1, p2), false otherwise. Returns false if s1/s2 is an invalid statement number.

9.11 PKBNext

- Overview: Extracts Next relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_NEXT createPKBNext(AST ast, NUM_OF_STMTS numberOfStmts)	Creates a PKB_NEXT.
BOOLEAN isNext(STMT_NUM s1, STMT_NUM s2)	Returns true if Next(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isNextS(STMT_NUM s1, STMT_NUM s2)	Returns true if Next*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.

9.12 PKBAffects

- Overview: Extracts Affects relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_AFFECTS createPKBAffect(AST ast, NUM_OF_ASSIGN numberOfAssignments)	Creates a PKB_AFFECTS.
BOOLEAN isAffects(STMT_NUM s1, STMT_NUM s2)	Returns true if Affects(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number, or not an assignment statement.
BOOLEAN isAffectsS(STMT_NUM s1, STMT_NUM s2)	Returns true if Affects*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number, or not an assignment statement.

9.13 PKBNextBip

- Overview: Extracts NextBip relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_NEXT_BIP createPKBNextBip(AST ast, NUM_OF_STMTS numberOfStmts)	Creates a PKB_NEXT_BIP.
BOOLEAN isNextBip(STMT_NUM s1, STMT_NUM s2)	Returns true if NextBip(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isNextBipS(STMT_NUM s1, STMT_NUM s2)	Returns true if NextBip*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.

9.14 PKBAffectsBip

- Overview: Extracts AffectsBip relations from the AST and answer queries regarding them
- Public Interface:

Operation header	Description
PKB_AFFECTS_BIP createPKBAffectsBip(AST ast, NUM_OF_STMTS numberOfStmts)	Creates a PKB_AFFECTS_BIP.
BOOLEAN isAffectsBip(STMT_NUM s1, STMT_NUM s2)	Returns true if isAffectsBip(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.
BOOLEAN isAffectsBipS(STMT_NUM s1, STMT_NUM s2)	Returns true if isAffectsBip*(s1, s2), false otherwise. Returns false if s1/s2 is an invalid statement number.

Appendix A - Parser Combinators

make_fun

```
make_fun : ParseFunc<StateT, X> → Parser<StateT, X>
```

Helper to reify a parsing function. Takes in a function with the appropriate parsing function signature, and returns a `Parser`.

This function is very useful as it allows us to define a parser without defining a new class and is used to construct most of the other parser combinators.

An example usage is to define other parser combinators, for example, `make_bind`.

```
template <...>
Parser<ParserStateT, OutType>
make_bind(Parser<ParserStateT, InType> &&parser, F fun) {
  return make_fun<ParserStateT, OutType>(
      // Provide a parsing function to make_fun
       [parser, fun](auto state) mutable -> std::optional<std::pair<OutType,
ParserStateT>> {
        // Apply the parser to the state to obtain
        // a possible value and new state
        auto first = parser(state);
        if (!first) {
          // If the parser failed, we fail as well
          return std::nullopt;
        }
        // Because the parser succeeded,
        // apply the user provided function
        // on the resulting value and state.
        return fun(std::move(first.value().first), first.value().second);
      });
}
```

As you can see, the implementation of most of the parser combinators is straightforwardly following their descriptions. If a parser combinator needs to "parse something, then parse something else", this is simply done by applying the first parser and then applying the second parser. This is the reason why both **writing** and **verifying** parser combinators are relatively easy.

From this point onwards, we will omit descriptions of their implementations because their implementations are identical to their descriptions.

make_non_owning

```
make_non_owning : (Parser<StateT, X> *) → Parser<StateT, X>
```

Helper to resolve recursive parsers. Takes in a pointer to a Parser, and dereferences it only when necessary to call the parsing function. This allows recursive parsers to be constructed, as each parser can store a pointer to the other.

This is implemented using make_fun, by simply reifying a parser lambda that dereferences the pointer and calls it.

make_lift

```
make_lift : (Parser<StateT, X>, (X → Y)) → Parser<StateT, Y>
```

"Lifts" a given function from a function on values to a function on parsers. Runs the given parser, and if it succeeds, runs the given function on the result of the parser. If the given parser fails, fail as well. This lets us transform the return type of parsers and thus is what lets us build the program AST and PQL QueryParseResult objects as required.

An example usage is:

```
make_lift<Program>(
     // The below expression is a parser for a program,
     // but its return value is not the correct AST node
     make_many(make_non_owning(&procedure)) << TOKEN_P(EOFTOK),
     [](std::vector<Procedure> in) {
       // This function converts the syntax of a program
       // i.e. a list of procedures
       // into the appropriate AST node (i.e. Program)
       return Program(std::move(in));
     });
```

Taking in a parser and a function, if the parser succeeds, it would have produced a vector of Procedures. The function then takes this vector and constructs a Program AST node from it. This lets us write the parser for Programs in terms of the parser for Procedures and Tokens.

make_bind

```
make_bind : (Parser<StateT, X>,
             ((X, StateT) → Optional<Pair<Y, StateT>))
            → Parser<StateT, Y>
```

The name comes from the "bind" operator of a Monad type. Runs the given parser, and if it succeeds, run the given function on both the result as well as the new state. If the given function subsequently returns a value, succeed with that value. If the given function returns a null optional, fail.

Notice that the given function in the second argument is basically a parsing function with the exception that an additional parameter of the earlier parser's result type is added. This is what makes the bind operation so general, as it allows us to chain parsers in a dependent fashion, with each subsequent parser depending on the result of previous parsers.

make_seq

```
make_seq : (Parser<StateT, X>, Parser<StateT, Y>, ...)
              → Parser<StateT, Tuple<X, Y, ...>>
```

Chains a (heterogeneous) list of parsers together, running each parser after the previous one succeeds. If any parser in the list fails, it backtracks all the way and fails as well. The result is stored in a tuple, which supports storing a list of values with heterogeneous types. This is used to implement concatenation in the BNF.

Because the types are heterogeneous, template magic is required to implement it.

An example usage for parsing conditional expressions:

```
make_seq(make_non_owning(&arith_expr),  // Parser for an expression
         make_alt(TOKEN_P(GT), ...),    // Parser for a comparison operator
         make_non_owning(&arith_expr))  // Parser for an expression
```

This returns a parser that parses an expression, followed by a comparison operator, followed by another expression, which corresponds to the non-terminal rel_expr, one of the cases of a cond_expr.

The type of this parser is `Parser<SIMPLEParserState, Tuple<ArithExpr, Token, ArithExpr>>`. To get a parser that properly returns a `CondExpr` instead of a tuple, see `make_lift`.

operator&&

```
(&&) : (Parser<StateT, X>, Parser<StateT, Y>)
              → Parser<StateT, Pair<X, Y>>
```

Operator overload for parsers. Similar to make_seq, it runs the left parser and then runs the right parser if the left one succeeded. The result is stored in a std::pair. This can be thought of as syntactic sugar for make_seq.

doperator<<

```
(<<) : (Parser<StateT, X>, Parser<StateT, Y>) → Parser<StateT, X>
```

Operator overload for parsers. Similar to operator&&, it takes in two parsers and runs them in sequence. Unlike operator&&, it only returns the result of the left parser. This can be thought of as syntactic sugar for make_seq, where the result of the right parser is discarded.

operator>>

```
(>>) : (Parser<StateT, X>, Parser<StateT, Y>) → Parser<StateT, Y>
```

Operator overload for parsers. Similar to operator&&, it takes in two parsers and runs them in sequence. Unlike operator&&, it only returns the result of the right parser. This can be thought of as syntactic sugar for make_seq, where the result of the left parser is discarded.

make_alt

```
make_alt : (Parser<StateT, X>, ...) → Parser<StateT, X>
```

Takes the alternative of a list of parsers, running each parser if the previous one failed, and succeeding after the first succeeding parser. This is used to implement disjunction in the BNF.

An example usage for parsing comparison operators:

```
make_alt(TOKEN_P(GT),   // Parser for >
         TOKEN_P(GTE),  // Parser for >=
         TOKEN_P(LT),   // Parser for <
         TOKEN_P(LTE),  // Parser for <=
         TOKEN_P(EQ),   // Parser for ==
         TOKEN_P(NEQ))  // Parser for !=
```

This returns a parser that parses exactly one of the required operators that are part of this parser. This is used in the parsing of the non-terminal rel_expr.

operator||

```
(||) : (Parser<StateT, X>, Parser<StateT, X>) → Parser<StateT, X>
```

Operator overload for parsers. Syntactic sugar for the make_alt of two parsers.

make_many

```
make_many : Parser<StateT, X> → Parser<StateT, Vector<X>>
```

Takes in a single parser. It runs the given parser zero or more times in sequence, until it fails, and returns each result in a vector. This is used to implement the Kleene star in the BNF.

An example usage for parsing lists of procedures:

```
Parser<...> procedure = ...;    // Define a parser for a single procedure

make_many(std::move(procedure)) // Parser for a list of procedures
```

This returns a parser that parses as many procedures as possible until an unexpected token is seen. The type of this parser is `Parser<..., Vector<Procedure>>`, and a `make_lift` can be used to convert a `Vector<Procedure>` into a `Program` AST node.

make_some

```
make_some : Parser<StateT, X> → Parser<StateT, Vector<X>>
```

Takes in a single parser. It runs the given parser one or more times in sequence, until it fails, and returns each result in a vector. This is used to implement + in the BNF.

An example usage for parsing lists of statements:

```
Parser<...> statement = ...;    // Define a parser for a single statement

make_many(std::move(statement)) // Parser for a list of statements
```

This returns a parser that parses as many statements as possible until an unexpected token is seen. The type of this parser is `Parser<..., Vector<Stmt>>`, and a `make_lift` can be used to convert a `Vector<Stmt>` into a `StmtLst` AST node.

Appendix B - Arc Consistency Algorithm

The Arc Consistency Algorithm #3 (AC3) is used by the QueryEvaluator to evaluate a given query. This is done by treating the query as a constraint satisfaction problem (CSP); each synonym (variable) has a domain of values it can potentially take, and the values within each domain can be restricted by the query's clauses (constraints) which dictate the relationships it has to other domains. AC3 operates on the CSP by using the constraints to continually reduce the possible set of values within the domain of each variable until all domains' values satisfy all the given constraints.

The activity diagram in Figure B.1 below illustrates the general flow of the algorithm:

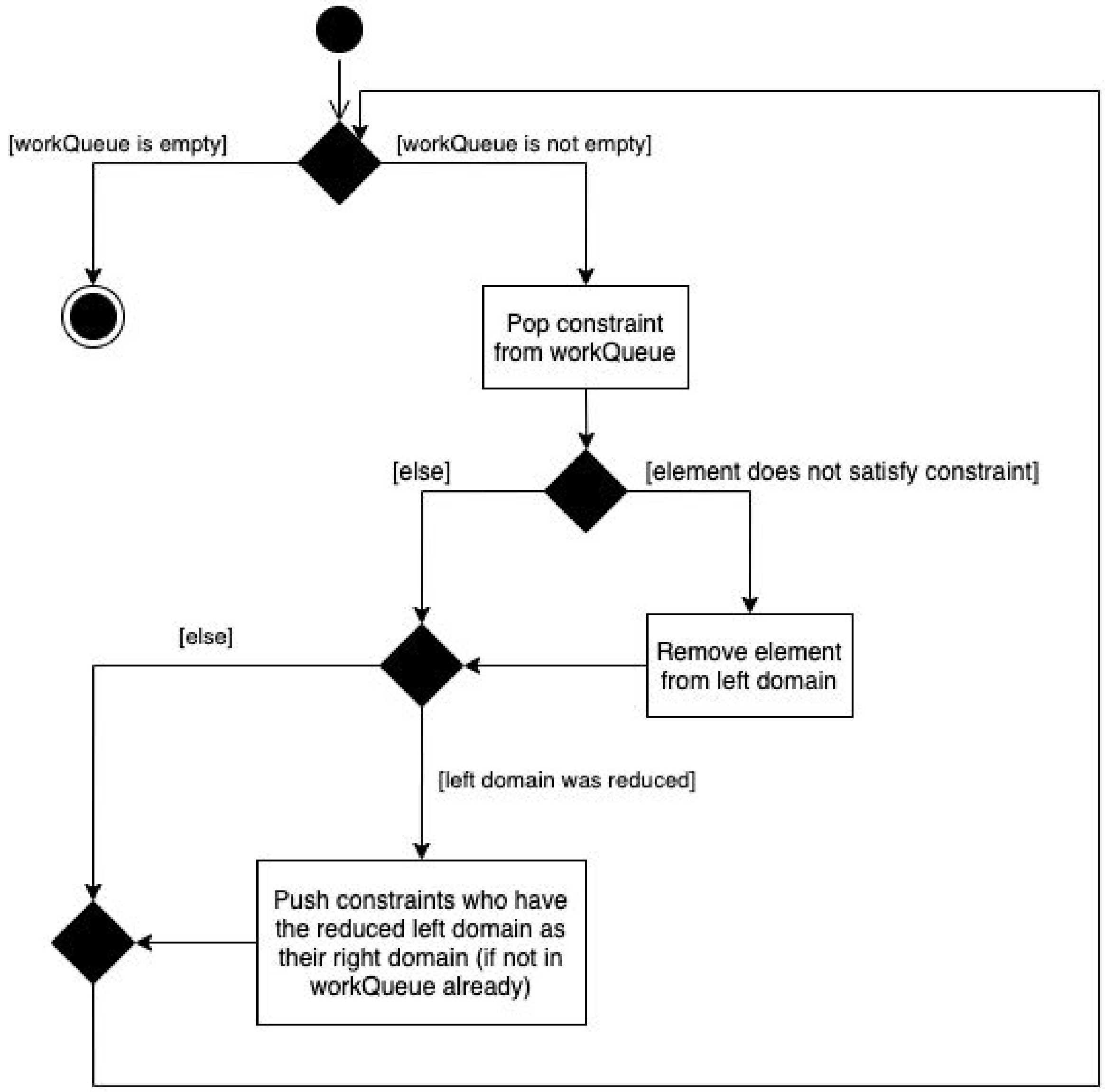

[workQueue is empty]
[workQueue is not empty]
Pop constraint from workQueue
[else]
[element does not satisfy constraint]
Remove element from left domain
[else]
[left domain was reduced]
Push constraints who have the reduced left domain as their right domain (if not in workQueue already)

Figure B.1: Activity Diagram for AC3 Algorithm

To further demonstrate how AC3 works for the evaluation of query clauses, we will use it to evaluate the following sample SIMPLE program and query:

```
procedure main {
1. read y;
2. while (x < 2) {
3.   x = x + 2;}
4. x = y;
5. print x;
6. while (x > 1) {
7.   x = x - 1;}
8. x = 11;}
9. while (x < 9) {
10.  x = x + 100;}

stmt s; assign a; constant c; while w;
Select s such that Follows (s, a) pattern w (c, _) with s.stmt# = c.value
```

From the query, we can identify the CSP variables and their initial domains by noting the synonyms declared.

Variable	**Domain**
`s`	`1, 2, 3, 4, 5, 6, 7, 8, 9, 10`
`a`	`3, 4, 7, 8, 10`
`c`	`1, 2, 9, 11, 100`
`w`	`2, 6, 9`

The constraints of the CSP can be translated from the query's clauses. For each of the relationships between two variables (synonyms), two constraints are created.

Note: ∀x reads as "for all x", ∃y reads as "there exists some y", and "|" reads as "such that".

Clause	**Constraint**
`Follows (s, a)`	`∀s∃a | Follows (s, a)`
	`∀a∃s | Follows (s, a)`
`pattern w (c, _)`	`∀w∃c | pattern w (c, _)`
	`∀c∃w | pattern w (c, _)`
`with s.stmt# = c.value`	`∀s∃c | s.stmt# = c.value`
	`∀c∃s | s.stmt# = c.value`

As can be seen in the table above, each constraint involves two domains. For example, the first constraint above (`∀s∃a | Follows (s, a)`) involves the domains `s` and `a`. For simplicity's sake, we shall refer to them as the left domain (`s`) and the right domain (`a`). With the variables, domains, and clauses above, we can construct a graph to represent the current status of the CSP.

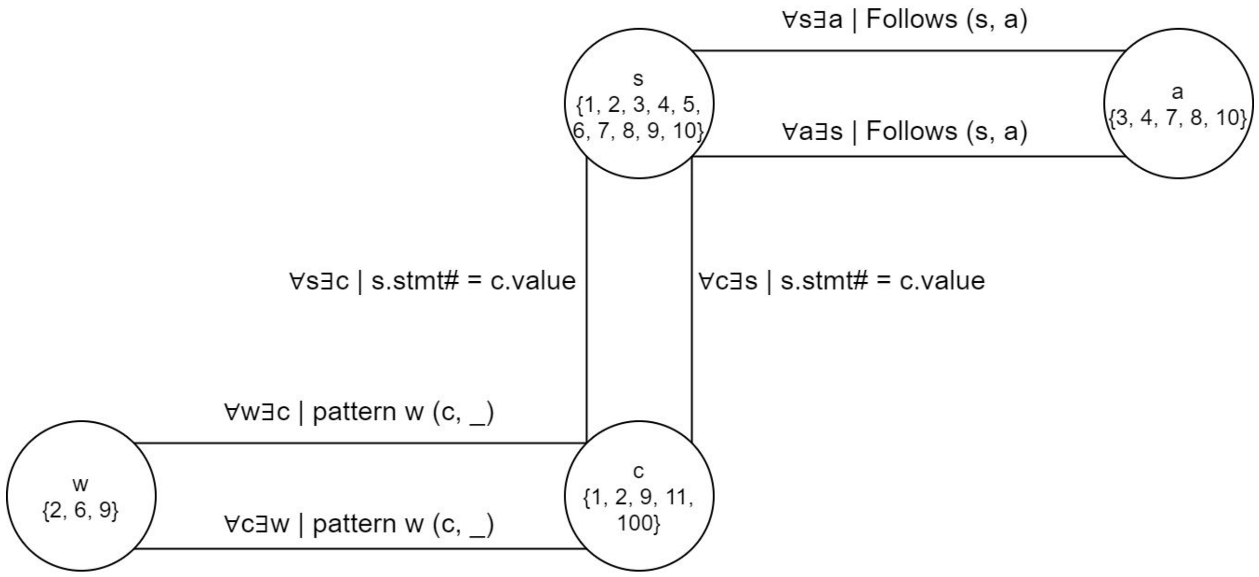


Figure B.2: Initial Constraint Graph for CSP

As seen in Figure B.2 above, the variables and their individual domains become the vertices of the graph while the constraints are represented by the edges between the variables. This graph contains all the data needed to begin AC3.

First, we create a queue `workqueue` containing all the constraints. The constraints are enqueued by the size of their left domains in ascending order. If multiple constraints have the same sizes for their left domains, they are ordered by the size of their right domains in ascending order. Thus, with the clauses above, `workqueue` would contain (in order):

Size		**Constraint**
Left	**Right**	
`w: 3`	`c: 5`	`∀w∃c	pattern w (c, _)`
`a: 5`	`s: 10`	`∀a∃s	Follows (s, a)`
`c: 5`	`w: 3`	`∀c∃w	pattern w (c, _)`
`c : 5`	`s: 10`	`∀c∃s	s.stmt# = c.value`
`s: 10`	`a: 5`	`∀s∃a	Follows (s, a)`
`s: 10`	`c: 5`	`∀s∃c	s.stmt# = c.value`

We then execute the following series of steps on `workqueue`:

1. Pop the constraint at the head. Here, it is: `∀w∃c | pattern w (c, _)`
2. Check if each element in the constraint's left domain (`w`) satisfies the constraint. If an element does not satisfy the constraint, it is removed from the left domain. In this example, we check if each element of `w` satisfies `pattern w (c, _)` for some value in the domain `c`:
    - 2.1. Element `2` satisfies the constraint as the domain `c` contains the element `2`.
    - 2.2. Element `6` does not satisfy the constraint as the domain `c` does not contain the element `6`. Element `6` is removed from the domain `w` (and the graph above is updated accordingly).
    - 2.3. Element `9` satisfies the constraint as the domain `c` contains the element `9`.
3. Now, we take into consideration whether the left domain was reduced. If so, we check for other constraints whose right domain is the reduced left domain. If these constraints are not currently in `workqueue`, they are pushed to the back of `workqueue`. In the case of the constraint we just evaluated, the left domain `w` was reduced and it appears as the right domain of the constraint `∀c∃w | pattern w (c, _)`. However, this constraint is already in `workqueue` and thus does not need to be enqueued again.

The above series of steps is repeated until `workqueue` is empty. This strategy is the meat of AC3 and will reduce the domains in the graph until only the possible values that satisfy all constraints remain such as in Figure B.3 below:

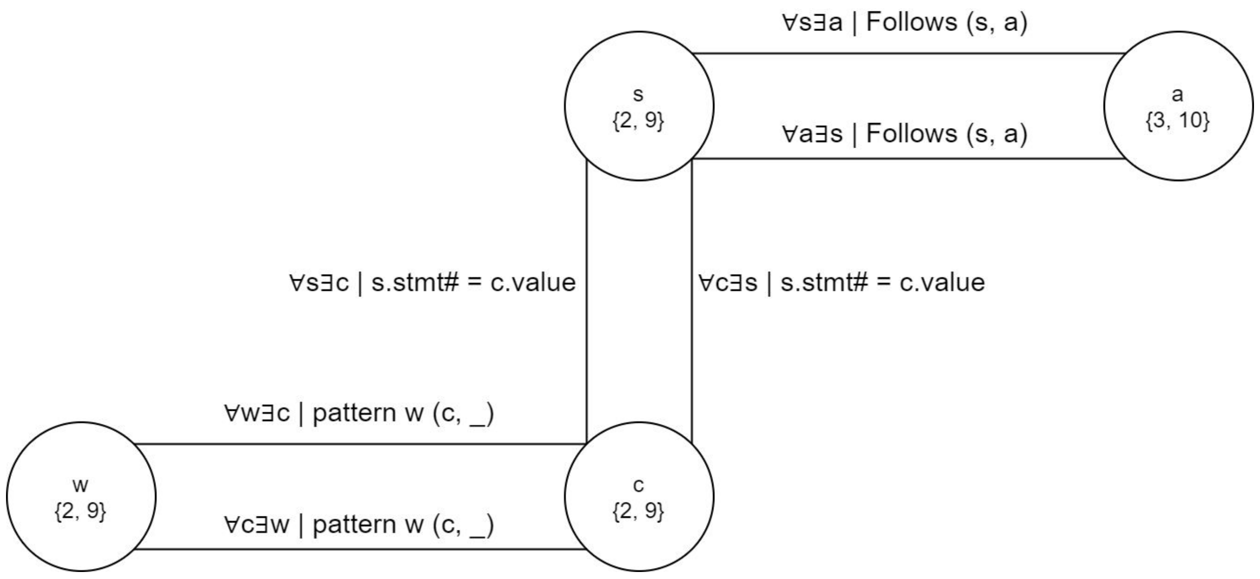


Figure B.3: Final Constraint Graph for CSP

With the domains reduced, enumerate the results. Taking the cartesian product of all reduced domains, we have 16 rows to check:

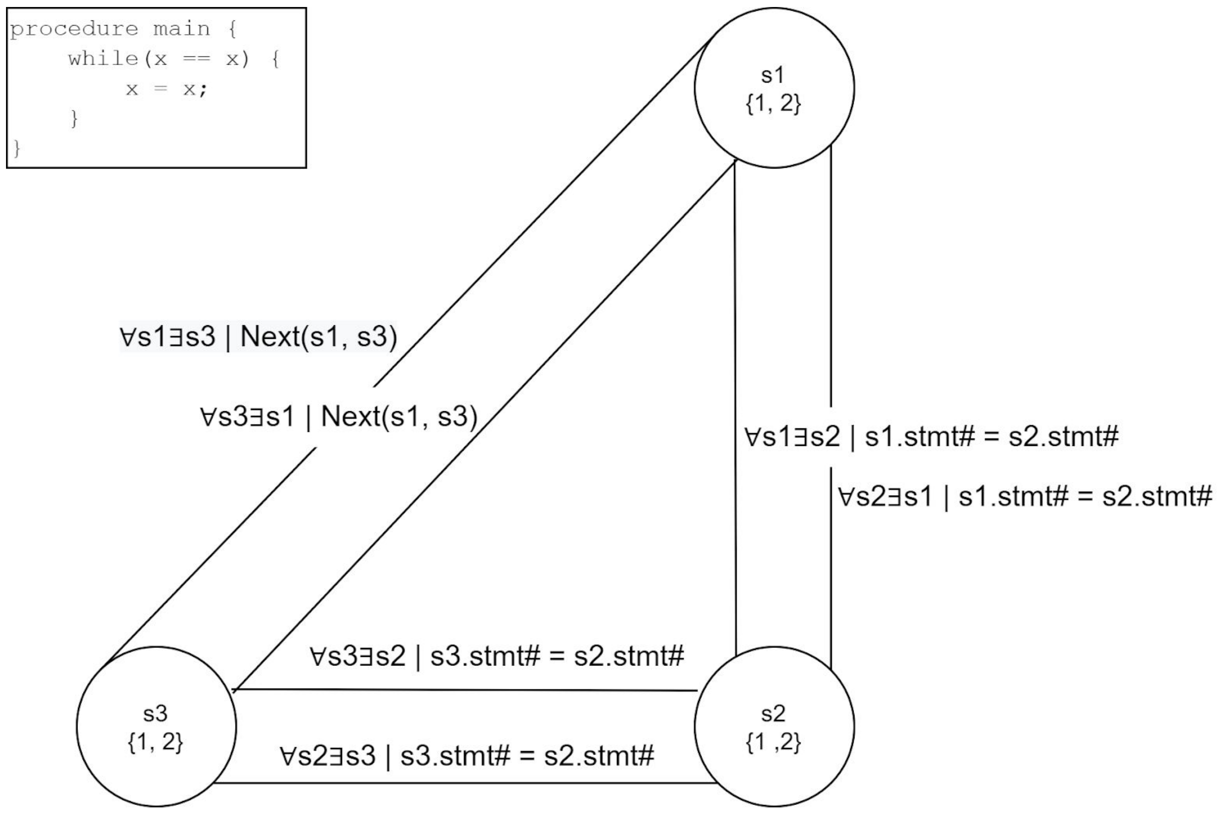
Synonyms				Valid?
w	c	s	a	
2	2	2	3	Y
2	2	2	10	Y
2	2	9	3	Y
2	2	9	10	Y
2	9	2	3	Y
2	9	2	10	Y
2	9	9	3	Y
2	9	9	10	Y
9	2	2	3	Y
9	2	2	10	Y
9	2	9	3	Y
9	2	9	10	Y
9	9	2	3	Y
9	9	2	10	Y
9	9	9	3	Y
9	9	9	10	Y

There is a valid assignment when s = 2, and when s = 9. Hence the enumeration retrieves the table:

s
2
9

In this case, all of the above assignments are valid against all constraints. We note that where originally we had 3 * 5 * 5 * 10 = 750 rows to check, the AC algorithm reduced our work to only 16 rows.

It might seem that taking the cartesian product and checking in this manner is redundant, but we could have contrived cases where the graph is arc-consistent but have no valid assignments such as in Figure B.4 below:



```
procedure main {
    while(x == x) {
        x = x;
    }
}
```

s1
{1, 2}
s2
{1 ,2}
s3
{1, 2}
∀s1∃s3 | Next(s1, s3)
∀s3∃s1 | Next(s1, s3)
∀s1∃s2 | s1.stmt# = s2.stmt#
∀s2∃s1 | s1.stmt# = s2.stmt#
∀s3∃s2 | s3.stmt# = s2.stmt#
∀s2∃s3 | s3.stmt# = s2.stmt#

Figure B.4: Arc-Consistent Graph with No Valid Assignments

Regardless of whatever synonym is selected, the full table join is empty. However, since AC only cares about pairwise constraints, the algorithm reports that the graph is arc-consistent.

Appendix C - Kameda’s Algorithm

Kameda’s algorithm is an algorithm used to check for reachability that can be done with O(N) pre-processing.

The algorithm can be used if the graph satisfies these properties:

- The graph is planer.
- The graph is acyclic.
- All 0-indegree and 0-outdegree vertices appear on the outer face, and it is possible to partition the boundary of that face into two parts such that all 0-indegree vertices appear on one part, and all 0-outdegree vertices appear on the other. In other words, all 0-indegree vertices are at the “top” of the graph, and all 0-outdegree vertices appear on the “bottom” of the graph.

Then, a depth-first-search is performed on the graph from left to right (the left child is recursed first). The nodes are then labelled recursively. We start with a global counter N, and the children of each node is first recursively labelled. Then, we label the current node with the counter value, and decrement counter by 1. Hence, we have a “left_label” of every node. We repeat this again, but instead, we perform depth-first-search from right to left instead to obtain a “right_label”.

A node can reach another node iff both its “left_label” and “right_label” is smaller than the target node, and this can be easily computed in O(1) time.

For more information: see https://en.wikipedia.org/wiki/Reachability#Kameda's_Algorithm

Appendix D - SIMPLE Source Programs for Testing

calculatepi.source.txt

```
procedure main {
  comment = estimatePiByEnteringRandomNumbers;
  comment = afterEnoughRandomNumbersAreInput;
  comment = itWillPrintAnApproximationOfPi;

  numIterations = 1000;

  call estimatePi;

  print piEstimate; }

procedure estimatePiInit {
  numInCircle = 0;
  numOutCircle = 0;

  call prngInit; }

procedure estimatePiIter {
  read randomInput;
  prngEntropy = randomInput;
  call prngAddEntropy;

  call prngGet;
  x = randomNumber;
  call prngGet;
  y = randomNumber;

  if (x * x + y * y > 1) then {
    numInCircle = numInCircle + 1; } else {
    numOutCircle = numOutCircle + 1; }
}

procedure estimatePiEnd {
  piEstimate = numInCircle / (numInCircle + numOutCircle) * 4;
}

procedure prngInit {
  prngstate = 2909182381;
  prnga = 3298409237;
  prngb = 54948993;
  prngc = 98239829; }

procedure prngAddEntropy {
  prngstate = prngstate + prngEntropy;
  call prngStep; }
```

```
procedure prngStep {
  prngstate = (prngstate * prnga + prngb) % prngc; }

procedure prngGet {
  randomNumber = prngstate / prngc;
  call prngStep; }

procedure estimatePi {
  call estimatePiInit;
  while (numIterations > 0) {
    numIterations = numIterations - 1;
    call estimatePiIter; }
  call estimatePiEnd; }
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	We test a simple join of two clauses (2.a.i)	Select <s1, s2, s3> such that Follows(s1, s2) and Parent(s2, s3)	5, 7, 8, 9, 11, 15, 17, 18, 19, 21, 23, 24
Test case 2	We test an “internal” join (2.a.ii)	Select pl1 such that Next*(pl1, pl1)	31, 32, 33
Test case 3	We test joins where the clause-graph is a 3-cycle (2.c.i)	Select <s1, pl2, s3> such that Next(s1, pl2) such that Next(pl2, s3) such that Next(s3, s1)	31 32 33, 32 33 31, 33 31 32
Test case 4	A projection test on a 4-cycle (3.b.i)	Select <p1, v1> such that Next*(a1, c1) with c1.procName = p1.procName such that Uses(p1, v1) pattern a1(v1, _)	estimatePi numIterations, prngAddEntropy prngEntropy, prngStep prngstate

edgecase_nest.source.txt

```
procedure cfgtest {
      x = 1;
      if (x == x) then {
            x = x; } else {
            x = x; }
      x = 5;
      while (x == x) {
            x = x; }
      x = 8;
      if (x == x) then {
            if (x == x) then {
                  x = x; } else {
                  x = x; } } else {
            if (x == x) then {
                  x = x; } else {
                  x = x; } }
      x = 16;
      if (x == x) then {
            while (x == x) {
                  x = x; } } else {
            while (x == x) {
                  x = x; } }
      x = 22;
      while (x == x) {
            if (x == x) then {
                  x = x; } else {
                  x = x; } }
      x = 27;
      while (x == x) {
            if (x == x) then {
                  x = x; } else {
                  x = x; }
            x = x; }
      x = 33;
}
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	We test if Next relation is valid when there are multiple CFG nodes going to one other node. (1.a.i)	Select s1 such that Next(s1, 16)	11, 12, 14, 15
Test case 2	There should be only one CFG node from a while block to the next statement after the block, regardless of what is in the while block. (1.a.i)	Select s1 such that Next(s1, 33)	28

basic_ifwhile_pattern.source.txt

```
procedure demo1 {
	x = x;
	if ((x9)>(9)) then {
		while (!((!((x5)!=(x6)))||(!((!((7)>=(7)))&&(!((x10)>(x5))))))) {
			if
(!((!(((x10)>=(5))&&(!((4)>=(x4)))))&&((!((3)>(x9)))&&(!((x10)<=(x9)))))) then {
				x = x;
			} else {
				x = x;
				x = x;
			}
		}
		while ((((1)%(x10))%((x7)/(x7)))>(7)) {
			x = x;
			x = x;
		}
	} else {
		x = x;
	}
	while ((!((x9)>=((x9)/(8))))||(!(((x9)/(x9))<(x4)))) {
		if (x == x) then {
			while
(!((!(((x1)<(x3))||(!((x6)<(x3)))))||(!((!((x1)!=(x4)))||(!((1)>(x1))))))) {
				x = x;
			}
			while (x == x) {
				x = x;
				x = x;
				if ((((3)-(6))-((x4)*(x7)))<=((x10)+((6)-(x5)))) then {
					x = x;
					x = x;
				} else {
					x = x;
					x = x;
					if ((x2)>=((x4)/((1)+(1)))) then {
						x = x;
					} else {
						x = x;
					}
				}
			}
		} else {
			x = x;
			x = x;
			x = x;
		}
		x = x;
		while (!(((!((3)<=(x3)))||((5)<(x2)))||(!((4)>(x5))))) {
			if (!((((1)-(x10))-((x6)+(x7)))!=((8)+((x6)*(x7))))) then {
```

```
                        if (!((x2)>(x3))) then {
                            if (!((x2)>(x3))) then {
                                x = x;
                            } else {
                                x = x;
                            }
                        } else {
                            x = x;
                            x = x;
                        }
                    } else {
                        while (x == x) {
                            x = x;
                        }
                        while (!((8)<=((x7)%(x9)))) {
                            x = x;
                            x = x;
                        }
                        x = x;
                    }
                }
            }
            if (x == x) then {
                if (!((((x2)-(x10))-((9)*(x6)))==((x9)-(x8)))) then {
                    x = x;
                } else {
                    x = x;
                    x = x;
                }
            } else {
                while (((!((x4)==(x6)))||((9)<=(x2)))||((!((6)>=(9)))&&(!((x1)!=(x8)))))
        {
                    x = x;
                    x = x;
                    while (x == x) {
                        x = x;
                        x = x;
                    }
                }
                x = x;
                while (!((((x2)-(x10))-((9)*(x6)))==((x9)-(x8)))) {
                    x = x;
                    x = x;
                }
            }
            x2 = 0;
        }
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	We test for wildcard arguments to while pattern clause (1.a.i)	Select w1 pattern w1(_,_)	3, 8, 12, 14, 16, 31, 39, 41, 50, 53, 57
Test case 2	We test for literal arguments to if pattern clause (1.a.i)	Select ifs1 pattern ifs1("x10",_,_) and ifs1("x4",_,_)	4, 19
Test case 3	Invalid arguments to if pattern clause (1.a.ii)	Select ifs1 pattern ifs1("1",_,_)	none

edgecase_with.source.txt

```
procedure t1 {
	print x3;
	print x8;
}
procedure t2 {
	read x4;
	print x5;
	read x6;
}
procedure t3 {
	read x1;
	print x1;
	read x2;
	print x2;
	read x3;

	print x4;
	read x5;
	print x6;
	read x7;

	x = 0 + 2 + 4;
	x = 5 + 6 + 9;
	x = 10 + 11 + 13 + 14;
	x = x;
	x = x;
	x = x;
}

procedure p1 {
	call p2;
	call p8;

	x = 21 + 22 + 23 + 24;
	x = 25 + 26 + 27 + 28 + 29;
	x = 30 + 31 + 32 + 33 + 34;

	x = 36 + 38 + 39;
	x = x;
	x = x;
	x = x;
	x = x;
}

procedure p2 {
	call p3;
}
```

```
procedure p3 {
      call p4;
}

procedure p4 {
      call p7;
}

procedure p5 {
      call p7;
}

procedure p6 {
      call p7;
}

procedure p7 {
      call p8;
}

procedure p8 {
      call p9;
}

procedure p9 {
      p8 = 0;
      print p7;
      read p6;
}
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	Comparing procedure name with variable name	Select v with p.procName = v.varName	p6, p7, p8
Test case 2	Comparing constant value with statement number	Select pr with c.value = pr.stmt#	2, 4, 9, 11, 13, 39
Test case 3	The above, joined with other valid with clauses	Select <r, pr, r.varName> with r.varName = v.varName and v.varName = pr.varName and pr.stmt# = c.value	12 4 x5, 8 9 x2, 3 11 x4, 5 13 x6

affects.source.txt

```
procedure affectstest {
    x = 1;
    y = x;
    y = x;

    x = 1;
    x = 2;
    y = x;

    x = 1;
    y = x;
    x = y;

    while (x == x) { x = x; }
    while (x == x) { x = x; x = 1; }

    x = 1;
    if (x == x) then { x = 2; } else { x = 3; }
    y = x;

    x = 1;
    if (x == x) then { x = 2; } else { y = 3; }
    y = x;

    x = 1;
    while (x == x) { x = 1; }
    y = x;

    x = 1;
    call modifiesx;
    y = x;

    x = 1;
    read x;
    y = x;

    x = 1;
    print x;
    y = x;

    x = 1;
    y = 1;
    z = 1;
    x = 1 + 2 * y / z - x;
}

procedure modifiesx {
    x = 1;
}
```

```
procedure diffprocnotaffected {
    y = x;
}
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	Check basic affects	Select BOOLEAN such that Affects(1,2)	TRUE
Test case 2	Check affects blocked by assignment	Select BOOLEAN such that Affects(4,6)	FALSE
Test case 3	Check self affects	Select BOOLEAN such that Affects(11,11)	TRUE
Test case 4	Check affects not blocked by if	Select BOOLEAN such that Affects(20,24)	TRUE

nextbip.source.txt

```
procedure proc1 {
    if (1 == 1) then {
        call proc2;
    } else {
        call proc3;
    }
    x = 1;
    call proc2;
}

procedure proc2 {
    while (1 == 1) {
        call proc3;
    }
    x = 1;
    call proc3;
}

procedure proc3 {
    x = 1;
}
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	Check for call stmt to next proc	Select BOOLEAN such that NextBip(2,6)	TRUE
Test case 2	Check call stmt in same proc not valid	Select BOOLEAN such that NextBip(2,4)	FALSE
Test case 3	Check NextBip* in different procedure	Select BOOLEAN such that NextBip*(1,10)	TRUE
Test case 4	Check while loop	Select BOOLEAN such that NextBip*(7,7)	TRUE

Sample random_program.txt

```
procedure proc1 {
	x2 = 
((x1)*((((x3)*((x2)*(x1)))-(((x1)%(x3))+((x2)+(84))))%((((356)*(800))-((622)/(x2)))%(
((x3)+(289))+((x1)/(36))))));
	x3 = 
(((((x2)+((840)+(x2)))*(875))+(x3))/(((((x1)-(x1))%((x2)+(x2)))*(x2))%((424)/(x3))));
	x2 = 
((((((511)+(x3))*((x3)-(x1)))%((961)*((x1)%(x1))))-((x1)-(((x2)*(637))+((985)+(423)))
))*((x2)+((((x3)%(x3))*((x3)/(x3)))*(((x2)%(947))%((706)+(x3))))));
}
procedure proc2 {
	x2 = ((((x3)-(((260)*(x1))*(x2)))-(x1))-((x2)+((((x1)/(x2))+(x3))/(509))));
	read x3;
}
procedure proc3 {
	call proc6;
	while 
(((((601)/(x3))-((x3)+(x3)))>=(x1))&&((!(((802)%(x2))!=((865)*(635))))||(((672)<=(215
))||(!((x1)>=(x1)))))) {
		x1 = 
(((x2)-(x2))/(((x1)*(((845)+(x2))-(6)))%((((520)/(x1))/(x3))+(x1))));
		while ((896)>=((x2)-(x1))) {
			x2 = (x3);
			x1 = 
((((((508)/(x1))+(476))%(((468)%(x3))%(x1)))-((((x3)+(x2))%((120)+(x2)))+(x3)))-((((2
09)/(x2))/(x2))*(((x1)+((x1)*(793)))%((462)/((x2)*(175))))));
		}
	}
}
procedure proc4 {
	x2 = (((x3)/(x2))+(((((717)/(638))*((x3)%(x3)))*((x2)*((x2)+(x3))))/(x1)));
	print x2;
	x3 = (x1);
}
procedure proc5 {
	read x3;
	call proc7;
	read x2;
}
procedure proc6 {
	print x1;
	while 
(!((!((!(((x2)+(x2))>=((x2)+(x1))))||(((795)-(x3))>((x3)*(x3)))))||((((x3)+(x3))!=(x2
))&&(!(((x2)%(286))<=(737)))))) {
		x1 = (x1);
		x3 = (x1);
	}
}
procedure proc7 {
      read x1;
      x1 =
((((((x1)+(x3))-((775)*(x1)))*((x3)+((571)+(x1))))*((((x2)*(x2))-((x2)%(x3)))+(((731)
*(x1))-((x1)+(x1)))))-(((((x3)/(60))+((x1)-(x1)))*(((344)+(x1))+((x2)-(391))))%((((x3
)*(x3))*(620))/((x2)*((x3)%(x2))))));
}
procedure proc8 {
      x3 =
((((((x1)*(x3))+((x1)-(x2)))/(((x2)/(x2))+(226)))-((((987)/(50))/((x3)*(760)))+(((413
)-(x3))+((851)+(x1)))))+((232)%((x2)*(((401)*(x3))+((x3)+(x3))))));
      if (((x2)!=(741))&&((533)==(((x1)*(x1))+(x2)))) then {
            x3 = (((749)-(720))*((15)/((x3)/((x1)%((67)%(648))))));
            x2 =
((((x1)+(395))/(((405)*((6)*(604)))*(((655)+(x2))-((746)-(x1)))))%(x2));
      } else {
            if (((x1)/(496))>=((882)+(((x1)+(x2))%(x3)))) then {
                  x3 =
((((((791)-(935))-((781)*(798)))-(x1))*((((119)-(647))*((x2)-(x2)))-(((x1)-(x3))*((59
1)+(870)))))*(x2));
                  if
(((((292)%(135))!=((581)+(x2)))&&(!(((x3)*(x2))<((x3)/(x1)))))||((x1)>=(((773)-(x2))-
((702)+(x3))))) then {
                        if
(!((!((x2)<(x1)))||(!((!((x2)>(x3)))&&(((x1)<(483))||((x2)>=(x1))))))) then {
                              read x1;
                              x1 = (150);
                        } else {
                              x1 =
(((226)-((x1)/(((x3)/(866))+((x3)-(x3)))))-(((((x3)-(x3))-(x2))/((x2)+(x3)))%(x1)));
                        }
                        print x2;
                  } else {
                        x2 =
(((x1)+((x1)%((x1)-(x2))))-(((((x1)/(649))/(78))-(303))*((956)/(((x2)+(x2))%(x3)))));
                        if
((!((((x1)+(x2))+((x3)%(955)))!=((x1)-((x2)-(x3)))))&&(!((((162)!=(837))&&(!((x3)>(89
3))))&&(!(((x2)<(x1))&&((x3)==(632))))))) then {
                              read x2;
                        } else {
                              x2 =
((x2)*(((((x3)+(948))*((x2)*(x3)))%(x2))*((((697)+(686))+((167)/(x3)))-(((x2)-(x2))+(
(568)/(x3))))));
                        }
                  }
            } else {
                  while
((!(((847)+((x1)+(x1)))<(((x2)*(x3))*((x2)%(x3)))))&&(!((!((!((x3)<=(583)))||((x3)==(
x2))))||(((x1)==(676))||(!((x1)<=(x1))))))) {
                        x3 =
((x3)*((x3)*((((x3)+(x1))-(531))%(((x2)*(x2))/(787)))));
                  }
```

```
                    x2 =
((((((x1)%(x3))+(668))%((x2)*((x2)*(x2))))+(((547)/((x3)/(156)))*(((x1)%(x2))%((x2)-(
x2)))))%(775));
                }
                x3 =
((((x1)-(x2))%(x3))-(((((x2)%(x3))/((547)%(486)))+(((x1)*(892))*(x2)))/((498)-((989)+
((276)+(x2))))));
        }
}
```

	Test Purpose	Required Test Inputs	Expected Test Results
Test case 1	Speed and check for failed assertions	Select <r1> such that Uses(s46, v54) such that Modifies(s46, v54) such that Modifies(s10, v54) such that Modifies(s10, v54) such that Modifies(s10, v54) such that Follows*(s54, r1)	-
Test case 2	Speed and check for failed assertions	Select <v16, c2, s47, s37> such that Parent*(s42, s47) such that Affects*(s42, s47) such that Affects*(s47, s42) such that Parent*(s47, s42) such that Parent(c2, s37) such that Next*(c2, s37) such that Affects(s37, c2) such that Affects*(s37, c2) such that Uses("estimatePiInit", v16)	-