INF2ATS

Analiza i testowanie systemów informatycznych

A Handbook

Stanisław Jarząbek

Table of contents:

1 Course and Handbook Overview

In this course, you learn software engineering principles and methods related to software design, implementation and testing. The course is project-based, so that you immediately learn how to effectively apply learned principles/method in practice. In teams, you implement a sizeable software system (over 10 KLOC), following an agile iterative development process. You learn fundamental principles and "best practices" of software engineering, architecture and component interface (API) design and specifications, analyzing and justifying design decisions; black box and white box testing, integration and system (acceptance) testing; writing test plans and reporting test results. Skills trained in the course: architecture design and specifications; component and subsystem interface design; planning and executing iterative development process; test planning, writing test cases, documenting test results; test tools. Teamwork and communication in a group - both in writing and discussion - are also an important objectives of the course.

In the first half of the course, you will work on problem understanding (requirements analysis), architectural design and prototyping (Part I and II of the Handbook). In the second half of the course, you will apply iterative development to build a system, working in a team with other students participating in the course. Course organization, schedule etc. will be discussed during lectures, please refer to the lecture notes for details.

This Handbook is your main text for the course. It explains in detail the problem you will be working on, and methods you should use to meet project requirements.

The project Handbook consists of four parts:

Part I: Software Requirements: Explains functional requirements and quality attributes for the software system you will implement.

Part II: Software Architecture and Design: Here you will find guidelines for the methods you should use in the project.

Part III: Software Process: In this part of the Handbook, look for everything that has to do with organizing a project work and a Software Development Life Cycle (SDLC) for the project.

Part IV: Techniques: Here you will find technical tips that should help you solve specific design and implementation problems.

Appendices summarize definitions introduced in different parts of the Handbook, for quick reference.

2 Succeeding in the course

First of all, you must keep in mind the objectives for this course, which are the following:

- Learn software principles and methods (design, implementation and testing) and how to apply them in practice.
- Develop the attitudes and abilities necessary to work effectively in a team.
- Develop communication and writing skills.
- Enhance development and problem-solving skills in areas of requirement analysis, architectural design, construction, testing and documentation.
- Learn how to evaluate design decisions at architecture and implementation levels.
- Enhance project planning skills.
- Apply and consolidate what you have learned in earlier programming courses.

A team project is an essential component of INF2ATS. Your performance in a team is an important element of course evaluation. All the students in the team share equal responsibility for creating team spirit, and making the team work as a whole. Should a problem arise, each student must be willing to work towards resolving the problem. If the problem proves too difficult to solve among yourselves, ask your supervisor for mediation; most problems can be resolved if addressed early. Each student should deliver work products according to the project plans set by the whole team.

Teamwork can be fun, but sometimes conflicts may arise. Students in a team have different personalities, skills, backgrounds and working styles. Naturally, different students contribute to the project in different ways, according to their skills, abilities and project plans set by the whole team.

ALL STUDENTS ARE EXPECTED TO HAVE EQUAL COMMITMENT AND PUT EQUAL EFFORT TO THE PROJECT

Project evaluation is a substantial part of the final grade. Your project will be evaluated based on these criteria:

1) Teamwork: The ability to work in a team.
2) Quality of architectural and detailed design decisions: Evaluation and selection of design decisions. What matters is not only the design decision itself, but also the process of arriving on design decisions, how you evaluate and argue about design choices, and clarity of documenting design decisions.
3) Reliability of SPA code: Conformance of SPA code to requirement specifications (checked during auto-testing of your SPA at the end of the course).
4) Quality of SPA code: implementation strategies; readability of code; coding standards and practices; conformance of implementation to architectural decisions (abstract APIs and information hiding).
5) Demonstrated maintainability, flexibility (design for change), reusability and scalability of design and code.
6) Quality of project report.
7) Quality of project presentation during final weeks of the course

Here are some tips to help you focus on essentials and not to "lose the forest for the trees":

1) Review "Required Program Quality Attributes" whenever you embark on a new project task.
2) "Just get a program running!" attitude will not work in this course. Take your time to design, test and document your programs.
3) We expect almost production quality software system. Ad hoc development will not yield the qualities we are looking for in your project. For that you need good design and rigorous testing. Good design will simplify testing, so invest in the design. Do not go for implementing much functionality if you cannot assure quality.
4) As you will develop a program incrementally, whatever you produce today, you will have to use and extend tomorrow. Ad hoc, poorly organized and documented program will cost you extra effort tomorrow. Eventually, it may lead to failure. Refine your documents to keep them up to date as you go through iterations.
5) Focus on reusability (develop once, reuse somewhere else), extensibility (design for change), and robustness (report an error if parts of the input are wrong). These are key factors in successful software development and will help you to easily evolve your program over time and adapt to changing requirements.
6) You work in a team. Work products (programs and documentation) you deliver are used by your teammates. To minimize miscommunication (you cannot eliminate it totally!), your work products must be easily understandable to others.
7) Interfaces among team members and among program components developed by different team members must be very well thought out and clear. Defining interfaces is "the heart" of architectural design – it is the key to the success of your project. Spend much time to get interfaces right – it will save your time as you progress though the project and will protect you against failure. Focus your weekly team meetings on discussing interfaces. We shall insist that you master the skill of defining interfaces and communicate with other teammates in terms of interfaces. This is one of the most important skills every good software designer must have.

8) Arrange team meetings on a weekly basis. Have a clear agenda for each meeting. List problems you want to discuss and be clear what you want to achieve at the end of the meeting.
9) Time management is a critical success factor for this course as you will need to juggle between this project and other coursework deadlines. Often, you will find yourself putting off this project in the face of other more urgent assignments' deadlines. However, we would like to remind you that the success of this project relies on the continuous and consistent effort of *every* team member *throughout* the course. Last minute work will only get you a poor grade. Hence, please weigh your priorities and manage your time accordingly.

3 The Policy on Project Work

We would like to inform you in advance of our expectations regarding academic conduct. Please ponder over the issues carefully so that you will not lose perspective in case you are tempted to violate the policy during the busy schedule of meeting your assignment deadlines.

For homework assignments and project assignments, you are permitted to discuss problem requirements and background material with anyone. However, the actual SOLUTIONS YOU HAND IN SHOULD BE YOUR OWN WORK – copying documentation or code (including copy & modify) is strictly prohibited. Throughout the course, you may discuss background material, problem requirements, approaches to problem solutions, and design with anyone, but you may not view any code or documentation written by anyone not in your team, including past students. Furthermore, you may not reveal your code and documentation to any students not in your team. The actual coding and documentation should be the work of your team only.

It happens, that academic misconduct cases come about because of poor judgment on the part of students who feel that they are incapable of completing an assignment. This feeling usually arises when you find yourself temporarily tired, stressed, or desperate. At times, we cannot avoid difficulties and problems. Overcoming them is a natural part of the learning process, it strengthens you. Please also bear in mind that the long-term consequences of an academic misconduct case will do much more damage to your career than the worst possible grade you can get in the course.

The course staff will not be happy to deal with any academic misconduct case, as this requires a huge amount of non-productive effort, and involves numerous university resources. However, for the long-term benefit and fairness of all students, we will prosecute every such case to the full extent provided for by the University regulation.

--- Part 1: Software Requirements ---

4 A Brief Problem Description

4.1 Motivation

Some companies spend as much as 80% of software budgets on software maintenance. During software maintenance, programmers spend almost 50% trying to understand a program. Therefore, methods and tools that can ease program understanding have a potential to substantially cut computing costs.

During program maintenance, programmers often try to locate codes relevant to the maintenance task at hand. Here are examples of questions programmers might ask to locate codes of interest:

1. Which procedures are called from procedure “Second”?
2. Which procedures call procedure “Second”?
3. Which variables have their values modified in procedure “Second”?
4. Find assignment statements where variable “x” appears on the left-hand side (LHS).
5. Find statements that contain sub-expression x * y + z.
6. Find three while-do loops nested one in another.
7. Find all program statements that modify variable “x” and use variable “y” at the same time.
8. Is there a control path from statement #20 to statement #620?
9. Which assignments that modify variable “x: affect value of “x” at statement #120?
10. Which program statements affect value of “x” at statement #120?
11. If I change value of “x” in statement #20, which other statements will be affected?
12. Find all assignments to variable “x” such that the value of “x” is subsequently reassigned a value in an assignment nested inside two loops.

To answer the above questions, a programmer may need to examine huge amount of code. Doing this by hand may be time consuming and error prone.

4.2 What Is an SPA and How Is It Used?

A **Static Program Analyzer (SPA for short)** is an interactive tool that automatically answers queries about programs. Of course, an SPA cannot answer all possible program queries. But the class of program queries that can be answered automatically is wide enough to make an SPA a useful tool. In your programming practice, you might have used a cross-referencing tool. An SPA is an enhanced version of a cross-referencing tool. In this project, you will design and implement an SPA for a simple source language.

The following scenario describes how an SPA is used by programmers:

1. John, a programmer, is given a task to fix an error in a program.

2. John feeds the program into SPA for automated analysis. The SPA parses a program into the internal representation stored in a Program Knowledge Base (PKB).

3. Now, John can start using SPA to help him find program statements that cause the error. John repeatedly enters queries to the SPA. The SPA evaluates queries and displays results. John analyzes query results and examines related sections of the program trying to locate the source of the error.

4. John finds program statement(s) responsible for an error. Now he is ready to modify the program to fix the error. Before that, John can ask the SPA more queries to examine a possible unwanted ripple effect of changes he intends to do.

From a programmer’s point of view, there are three use cases: source program entering and automated analysis, query entering and processing, and viewing query results.

4.3 How Does an SPA Work?

In order to answer program queries, an SPA must first analyze a source program and extract relevant program design abstractions. Program design abstractions that are useful in answering queries typically include an Abstract Syntax Tree (AST), a program Control Flow Graph (CFG) and cross-reference lists indicating usage of program variables, procedure invocations, etc. An *SPA Front-End* (Figure 1) parses a source program, extracts program design abstractions and stores them in a Program Knowledge Base (PKB).

Now, we need to provide a programmer with means to ask questions about programs. While using plain English would be simple for programmers, it would be very difficult for SPAs. Therefore, as a workable compromise, we define a semi-formal Program Query Language (*PQL* for short) for a programmer to formulate program queries. A *Query Processor* validates and evaluates queries. *Query Result Projector* displays query results for the programmer to view.

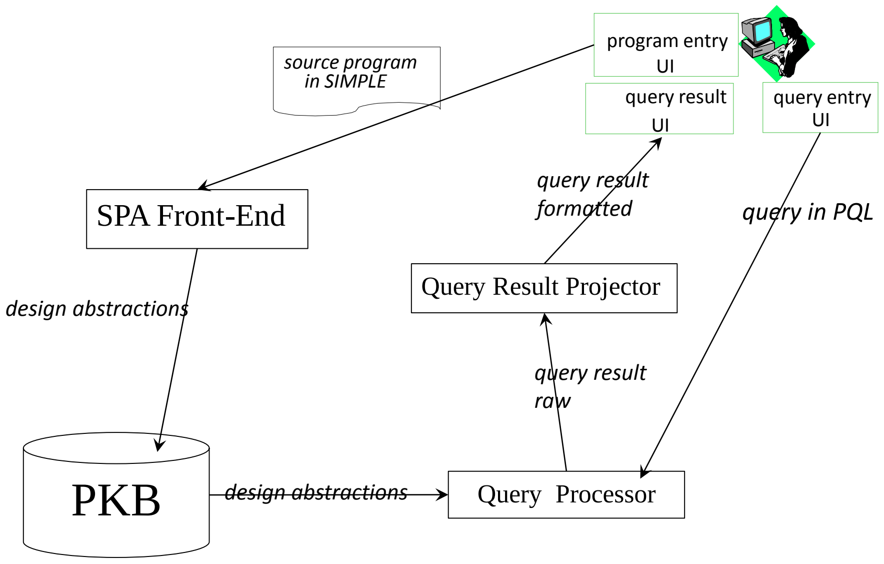


Figure 1. Logical components and operational scenario of a Static Program Analyzer

It should be easy for a programmer to write program queries. In particular, a programmer should be able to concentrate on the program itself rather than on the representation of program design abstractions in the PKB. For that reason, our *PQL* will allow a programmer to ask program queries in terms of *program design models* rather than in terms of *physical program representations* stored in the PKB. *PQL* is similar to SQL, but it is a higher level language than SQL. For example, query:

Which procedures are called from procedure "Second"? "

can be written in *PQL* as follows:

Q1. procedure p;
Select p **such that** Calls ("Second", p)

To answer queries, the *Query Processor* (Figure 1) will map program design entities referenced in a query (such as *procedure*) and their relationships (such as *Calls*) into the program design abstractions stored in the PKB in the "raw form". Next, Query Evaluator will interpret query conditions in **such that**, **with** and other query clauses to extract from the PKB program query results, i.e., program design entities that match the conditions.

5 Source Language *SIMPLE*

Your SPA will analyze programs written in a source language called *SIMPLE*. *SIMPLE* was designed for the purpose of experimenting with SPA techniques, not as a language for solving real programming problems. The language contains the minimum number of constructs to serve that purpose. The example below depicts a program structure and language constructs in *SIMPLE*. Program lines in procedure "Second" are numbered for ease of referencing in examples.

```
procedure First {
   x = 2;
   z = 3;
  call Second; }
procedure Second {
1.  x = 0;
2.  i = 5;
3.  while i  {
4.          x = x + 2 * y;
5.         call Third;
6.          i = i  - 1; }
7.  if x then {
8.          x = x + 1; }
    else {
9.          z = 1; }
10.  z = z + x + i;
11.  y = z + 2;
12.  x = x * y + z; }
procedure Third {
    z = 5;
    v = z;  }
```

Figure 2. Sample program in *SIMPLE*

Here is a summary of the language rules for *SIMPLE*:

1) A **program** consists of one or more **procedures**. Program execution starts by calling the first procedure in a program. By convention, the name of the first procedure is also the name of the whole program.

2) Procedures have no parameters, and cannot be nested in each other or called recursively.

3) A procedure consists of a **non-empty** list of **statements**.

4) **Variables** have unique names and global scope. Variables require no declarations.

5) There are four **types of statements**, namely procedure call, assignment, while loop and if-then-else. Decisions in while and if statements are made based on the value of a control variable: TRUE for values different than 0 and FALSE for 0. All the variables are of integer type, and are initialized to 0.

6) **Operator** * has higher priority than + and -. Operators + and - have equal priority. Operators are left-associative, meaning the operations of the same priority are grouped from the left to the right.

Below is a concrete syntax grammar for *SIMPLE*. Lexical tokens are written in capital letters (e.g., LETTER, NAME). Keywords are between apostrophes (e.g., 'procedure'). Non-terminals are in small letters.

Concrete Syntax Grammar for *SIMPLE*

Meta symbols:

```
a*      -- repetition of a, 0 or more times
a+      -- repetition of a, 1 or more times
a | b   -- a or b
brackets ( and ) are used for grouping
```

Lexical tokens:

LETTER : A-Z | a-z -- capital or small letter
DIGIT : 0-9
NAME : LETTER (LETTER | DIGIT)* -- procedure names and variables are strings of letters, and digits, starting with a letter
INTEGER : DIGIT+ -- constants are sequences of digits

Grammar rules:

```
program : procedure+
procedure : ‘procedure’ proc_name ‘{‘ stmtLst ‘}’
stmtLst : stmt+
stmt : call  |  while | if | assign
call : ‘call’ proc_name ‘;’
while : ‘while’ var_name ‘{‘ stmtLst ‘}’
if : ‘if’ var_name ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’
assign : var_name ‘=’ expr ‘;’
expr : expr ‘+’ term | expr ‘-’ term | term
term : term ‘*’ factor | factor
factor : var_name | const_value | ‘(’ expr ‘)’
var_name : NAME
proc_name : NAME
const_value : INTEGER
```

Other rules:

1. It is an error to have two procedures with the same name, as well as a call to a non-existing procedure. Recursive calls are not allowed, either.
2. Spaces (including multiple spaces or no spaces) can be used freely, as long as it does not lead to ambiguity, and your tokenizer should deal with that. For example, your tokenizer should recognize three tokens “x”, “+” and “y” in any of the following three character streams:

   x+y,   x + y,   x   +y
3. SIMPLE programs are always in one-statement-per-line format as in examples.
4. You can make your own assumptions about any other details that have not been specified above – such as case-sensitivity of identifiers/keywords, whether a procedure name may be the same as a variable name or not, etc. Make sure that these assumptions you make are explicitly discussed in your documentation.

6 Program Design Abstractions for *SIMPLE*

The SPA front-end will parse source programs, derive program design abstractions from sources and store them in the PKB. *PQL* query subsystem will then consult the PKB to validate and evaluate program queries. In this section, we address two important issues:

1. How do we specify program design abstractions to be stored in the PKB?
2. How do we refer to program design abstractions in *PQL* queries?

The models described in this section provide the answer to the above questions. Why use models rather than just describe the physical contents of the PKB? Data structures in the PKB may be complex. Writing queries directly in terms of data structures would be unnecessarily complicated while, as we already noted, *PQL* should be easy to use for programmers.

Program design abstractions described in this section are "conceptual" in the sense that they make no assumptions about their physical data representation in the PKB. At the same time, models are sufficient to formulate program queries in a precise but simple and natural way. This is the most important reason for modeling program design abstractions. There are more reasons that make conceptual modeling worthwhile, but as these reasons have to do with the design of a query processing subsystem, we shall defer their discussion until the later sections.

Program design abstraction models are defined in terms of program design entities (such as *procedure*, *variable* or *assignment*), entity relationships (such as *Calls* or *Follows*) and entity attributes (such as *procedure.procName* or *variable.varName*). For each model, you will find a graphical definition (as an UML class diagram) and equivalent textual definition.

6.1 Procedure Calls, Modifies and Uses Relationships

In the figure below, we see calling relationships among procedures modeled as a UML class diagram.

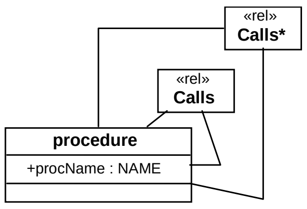


Figure 3. Calls and Calls* relationship

Program design entities are represented as classes. Relationships among program design entities are represented by class stereotypes <<rel>>. The meaning of relationships is explained below. For any procedures p and q:

Relationship **Calls** (p, q) holds if procedure p directly calls q.

Calls* (p, q) holds if procedure p calls directly or indirectly q, that is:

Calls* (p, q) if:

Calls (p, q) or

Calls (p, p1) and Calls* (p1, q) for some procedure p1.

Examples using sample program of Figure 2:
Calls ("First", "Second"), Calls ("Second", "Third"), and
Calls* ("First", "Second"), Calls* ("Second", " Third"), Calls* ("First", "Third").

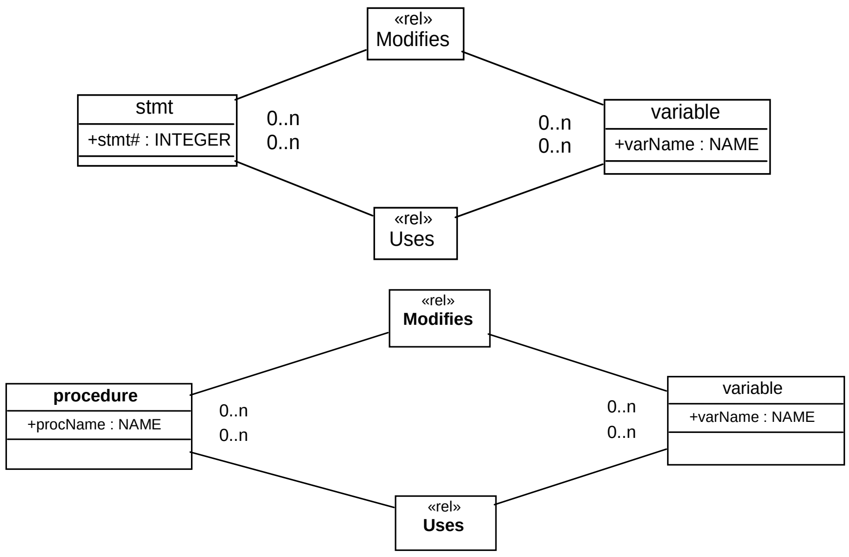


Figure 4. Modifies and Uses for statements and procedures

Relationship **Modifies** is defined as follows:

1. For an assignment a, Modifies (a, v) holds if variable v appears on the left-hand side of a.
2. For a container statement s (i.e., 'if' or 'while'), Modifies (s, v) holds if there is a statement s1 in the container such that Modifies (s1, v) holds.
3. For a procedure p, Modifies (p, v) holds if there is a statement s in p or in a procedure called (directly or indirectly) from p such that Modifies (s, v) holds.
4. For a procedure call statement s 'call p' Modifies (s, v) is defined in the same way as Modifies (p, v).

The above definition implies that in the example below we have: Modifies (1, "x"), Modifies (2,"x"), Modifies (1,"y"), Modifies (3, "y"), Modifies ("p", "x"), and Modifies ("p", "y").

```
procedure p {
1.      if x then {
2.          x = 10; }
        else {
3.          y = 20; } }
```

Relationship **Uses** is defined as follows:

1. For an assignment a, Uses (a, v) holds if variable v appears on the right-hand side of a.
2. For a container statement s (i.e., 'if' or 'while'), Uses (s, v) holds if v is a control variable of s (such as 'while v' or 'if v'), or there is a statement s1 in the container such that Uses (s1, v) holds.
3. For a procedure p, Uses (p, v) holds if there is statement s in p or in a procedure called (directly or indirectly) from p such that Uses (s, v) holds.
4. For a procedure call statement s 'call p' Uses (s, v) is defined in the same way as Uses (p, v).

For example, using sample program of Figure 2 we have:

Modifies (1, "x"), Modifies (2, "i"), Modifies (6, "i")

Modifies (3, "x"), Modifies (3, "z"), Modifies (3, "i")

Uses (4, "x"), Uses (4, "y")

Notice that if a number refers to statement s that is a procedure call, then Modifies(s, v) holds for any variable v modified in the called procedure (or in any procedure called directly or indirectly from that procedure). Likewise for relationship Uses. Also, if a number refers to a container statement s (i.e., ‘while’ or ‘if’ statement), then Modifies(s, v) holds for any variable modified by any statement in the container s. Likewise for relationship Uses.

More examples using the sample program of Figure 2 (we refer to procedures and variables via their names given as strings):

Modifies (“First”, “x”), Modifies (“First”, “z”)
Modifies (“Second”, “ i”)
Modifies (“First”, “i)
Modifies (“Third”, “v”), Modifies (“Second”, “v”), Modifies (“First”, “v”),
Uses (“Second”, “ i”), Uses (“Second”, “x”), Uses (“Second”, “y”), Uses (“Second”, “z”),
Uses (“First”, “i”), Uses (“First”, “x”), Uses (“First”, “y”) , and Uses (“First”, “z”),

Your SPA front-end will derive program design abstractions specified by the above model from source programs and store them in the PKB. We are not talking about the data representation for program design abstractions yet.

6.2 A Grammar Model of Abstract Syntax of *SIMPLE*

Your SPA front-end will build an abstract syntax tree (AST) for a source program. Nodes in the AST are instances of design entities such as ‘assign’, ‘expr’ or ‘while’. The following rules model the abstract syntax structure of programs in *SIMPLE*:

```
Meta symbols: a+ means a list of 1 or more a’s; ‘|’ means or

Lexical tokens:
LETTER : A-Z | a-z -- capital or small letter
DIGIT : 0-9
NAME : LETTER (LETTER | DIGIT)*
INTEGER : DIGIT+

Grammar rules:
program : procedure+
procedure : stmtLst
stmtLst : stmt+
stmt : assign | call | while | if
assign : variable  expr
expr : plus | minus | times | ref
plus : expr  expr
minus : expr  expr
times : expr  expr
ref : variable | constant
while:  variable  stmtLst
if : variable  stmtLst  stmtLst

Attributes and attribute value types:
procedure.procName, call.procName, variable.varName :  NAME
constant.value : INTEGER
stmt.stmt# : INTEGER
```

Figure 5. A grammar model of abstract syntax structure for *SIMPLE*

You may like the following graphical model of abstract syntax of *SIMPLE* (Figure 6). Textual definition of abstract syntax (Figure 5) is equivalent to graphical definition (Figure 6).

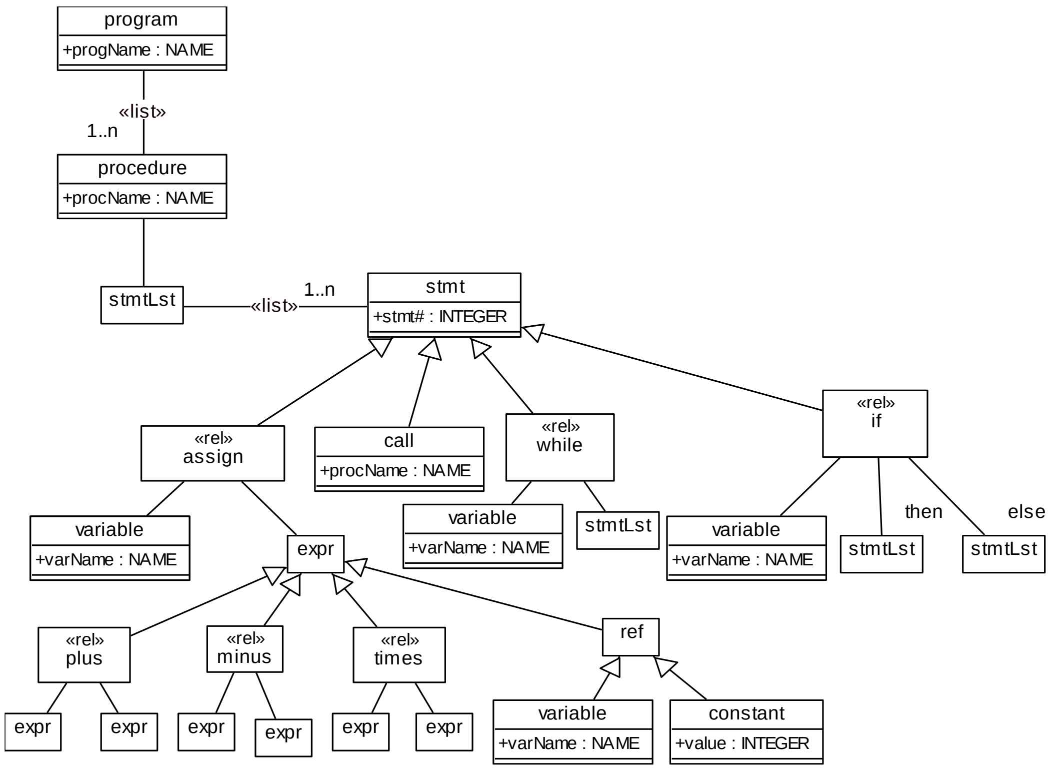


Figure 6. Abstract syntax grammar rules for *SIMPLE* in a diagram form

A diagram of Figure 7 depicts abstract syntax tree (AST) nodes and important relationships among them. The diagram shows a partial AST for program First (Figure 2).

AST nodes are labeled with values (procedure/variable names, constants, placed before ":") and syntactic types (placed after ':'). For example, node Second:call represents a call to the procedure Second, and x:variable a reference to variable x.

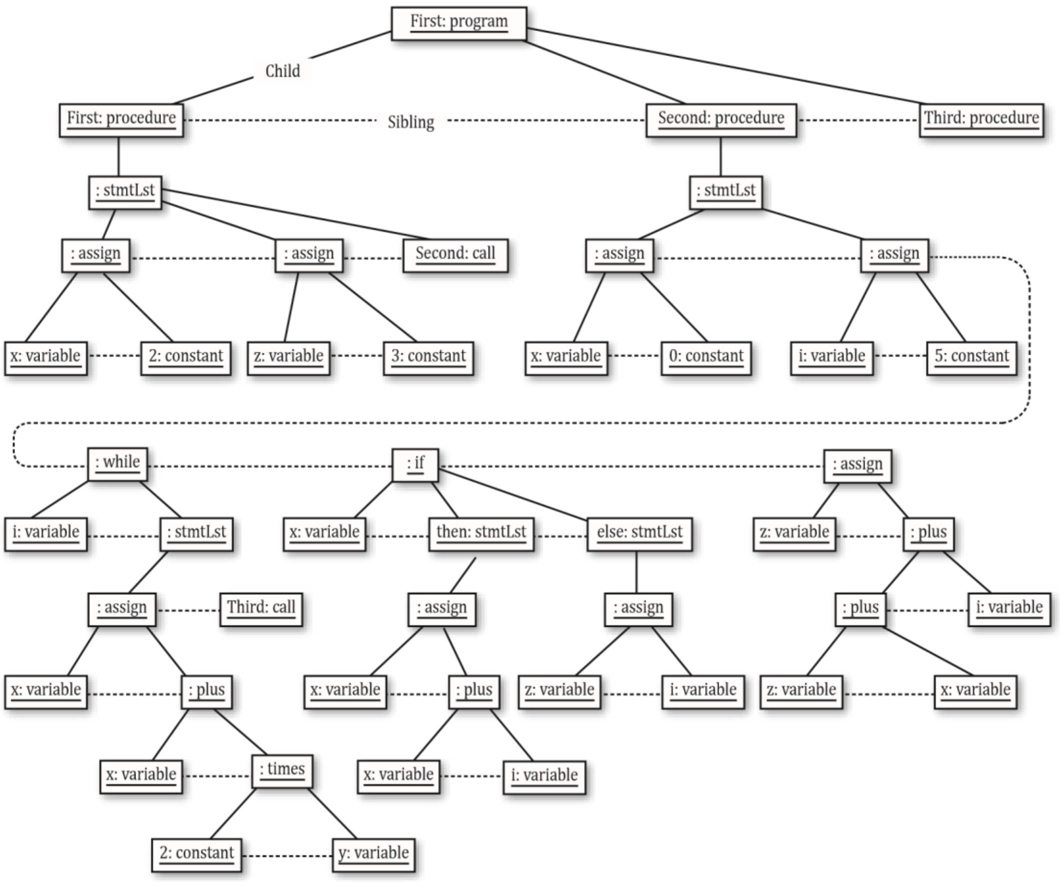


Figure 7. A partial AST for program First

Observe that the AST is built strictly according to abstract syntax grammar rules for *SIMPLE*: Each non-leaf node corresponds to a syntactic type on the left-hand side of some grammar rule, and its children nodes correspond to the syntactic types on the right-hand side of that grammar rule. For any language unambiguously defined (such as *SIMPLE*), each well-formed program corresponds to one and only one AST.

Let tn, tn_1 and tn_2 be any nodes of AST. The following are some of the important relationships that define organization of nodes in AST:

Node tn_1 is a child of tn if tn_1 appears directly below tn in AST. Child relationship is shown as a solid line in Figure 7.

We refer to the left-most child of node tn as the first child of tn.

Node tn_2 is the right sibling of tn_1 if tn_1 and tn_2 are children of the same node, and tn_2 appears directly on the right-hand side of tn_1. Sibling relationship is shown as a dotted line in Figure 7.

The AST of Figure 7 and the above relationships describe general concepts of AST. They do not prescribe or suggest any implementation of AST, which will be up to you to decide.

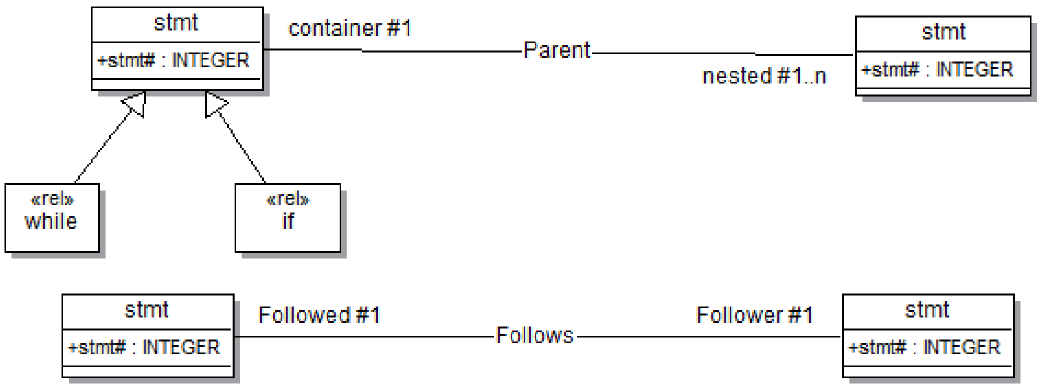


Figure 8. Relationships Parent and Follows

Relationships 'Parent' and 'Follows' describe the nesting structures of program statements within a procedure. Relationships 'Parent' and 'Follows' are implicitly defined in the AST.

For any two statements s1 and s2, the relationship **Parent** (s1, s2) holds if s2 is directly nested in s1. Therefore, s1 must be a 'container' statement. In *SIMPLE* there are two containers, namely 'while' and 'if'. In terms of AST, Parent (s1, s2) holds if s2 is a direct child of stmtLst which in turn is a direct child of the AST node labeled with the container name ('while' or 'if').

In examples, we refer to statements via their statement numbers, e.g., Parent (3, 4). As Parent relationship is defined for program statements, statement number '3' refers to the whole while statement, including lines 3-6, rather than to the program line "while i (" only. Similarly, statement number '7' refers to the whole if statement includes lines 7-9.

For example, in procedure Second (Figure 2) the following relationships hold: Parent (3, 4), Parent (3,5), Parent (3,6), Parent (7,8), Parent (7, 9), etc.

Relationship **Parent*** is the transitive closure of relationship 'Parent', i.e.,

Parent* (s1, s2) if:

Parent (s1, s2) or

Parent (s1, s) and Parent* (s, s2) for some statement s.

For any two statements s1 and s2 relationship **Follows** (s1, s2) holds if s2 appears in program text directly after s1 at the same nesting level, and s1 and s2 belong to the same statement list (stmtLst). In terms of AST, Follows (s1, s2) holds if s2 is the direct right sibling of s1.

Follows* is the transitive closure of Follows, i.e.,

Follows* (s1, s2) if:

Follows (s1, s2) or

Follows (s1, s) and Follows* (s, s2) for some statement s.

For example, in procedure Second, the following relationships hold: Follows (1, 2), Follows (2, 3), Follows (3, 7), Follows (4, 5), Follows (7, 10), Follows* (1, 2), Follows* (1, 3), Follows* (1, 7), Follows* (1, 12), Follows* (3, 12). Statement 6 is not followed by any statement. Notice that statement 9 does not follow statement 8. As in case of the Parent relationship, number '3' refers to the whole while statement and number '7' refers to the whole if statement.

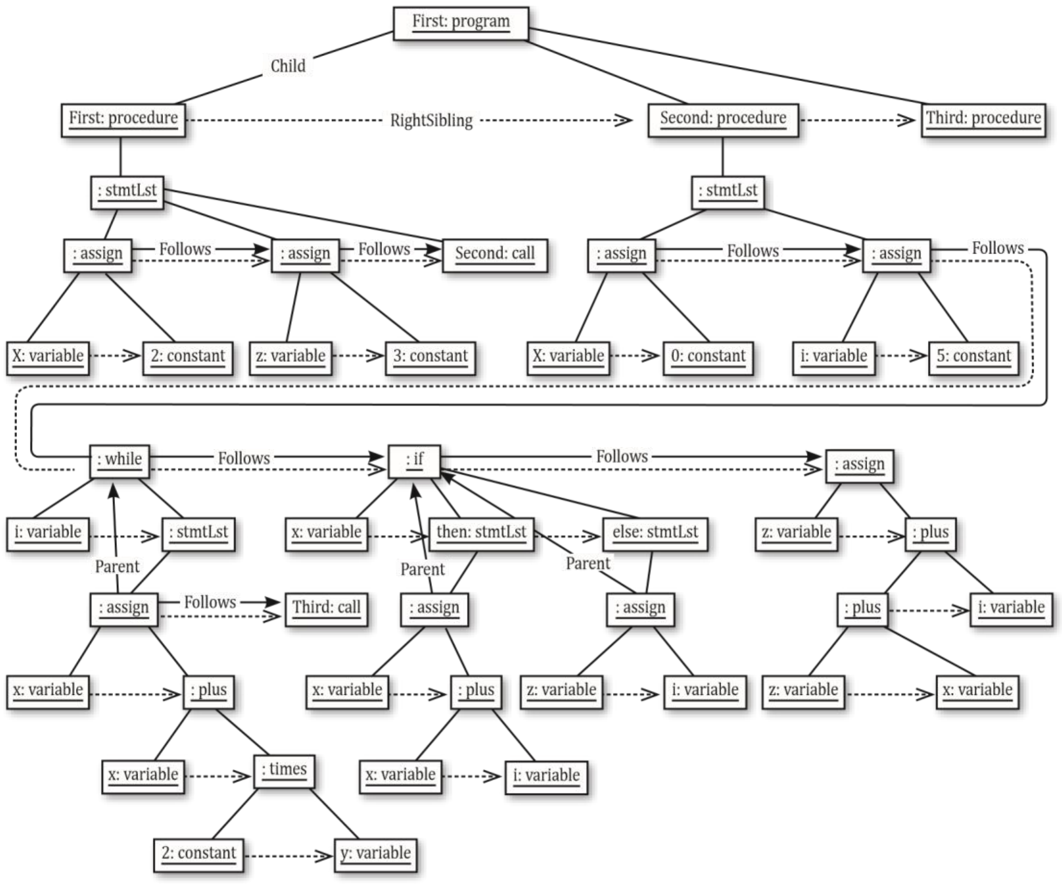


Figure 9. A partial AST for program First with links Follows and Parent

Figure 9. A partial AST for program First with links Follows and Parent

6.3 A Model of Program Control and Data Flow

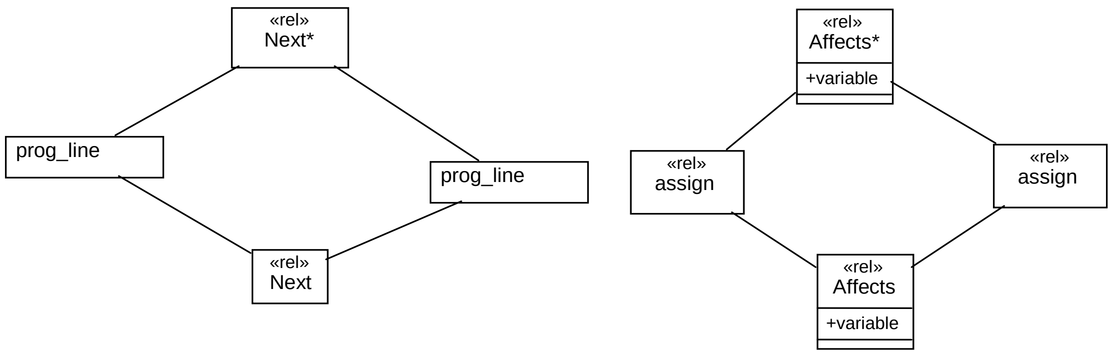


Figure 10. A model of program control and data flow

Let n_1 and n_2 be program lines.

Relationship **Next** (n_1, n_2) holds if n_1 and n_2 are in the same procedure, and n_2 can be executed immediately after n1 in some program execution sequence.

Note that we define control flow (Next) only in the scope of a procedure, not across procedures. Program lines belonging to two different procedures cannot be related by means of relationship Next.

In case of assignments or procedure calls, a line number also corresponds to a statement. But for container statements, line numbers refer to the "header" of the container rather than the whole statement as it was the case before. For example, line number ‘3’ refers to “while i {“ and line number ‘7’ refers to “if x then {“.

For example, in procedure Second we have:

Next (1, 2), Next (2, 3), Next (3, 4), Next (4, 5), Next (5, 6), Next (6, 3), Next (3, 7),

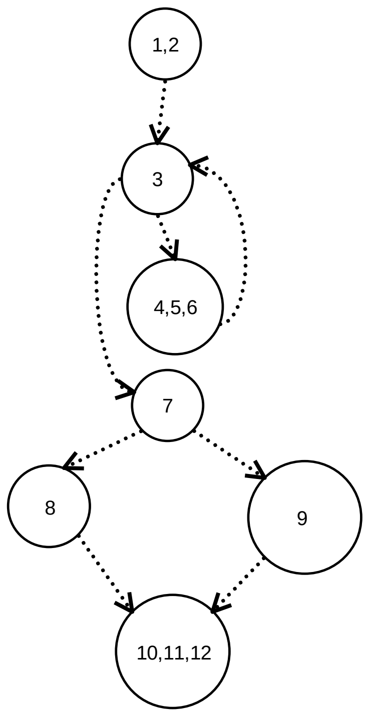
Next (7, 8), Next (7, 9), Next (8, 10), Next (9, 10), Next (10, 11) and Next (11, 12)

A **control flow path** is any sequence of program lines (n_1, …, n_k)
such that Next (n_i, n_{i+1}) for i = 1, …, k-1

(3, 4, 5, 6, 3, 7) is an example of a control flow path in procedure Second.

Based on the relationship ‘Next’, we can define a Control Flow Graph (CFG) in a program. CFG is a compact and intuitive representation of all control flow paths in a program. Nodes in the CFG are so-called basic blocks. A basic block contains program line numbers that are known to be executed one by one, in sequence. That is, a decision point in “while” or “f” always marks the end of a basic block. As program lines within a node are known to execute sequentially, we only need to explicitly show control flows among basic blocks.

```
procedure Second {
1.  x = 0;
2.  i = 5;
3.  while i  {
4.          x = x + 2 * y;
5.         call Third;
6.          i = i  - 1; }
7.  if x then {
8.          x = x + 1; }
    else {
9.          z = 1; }
10.  z = z + x + i;
11.  y = z + 2;
12.  x = x * y + z; }
```



The relationship **Next*** is the transitive closure of relationship ‘Next’:

Next* (n_1, n_k) if

Next (n_1, n_k) or

Next (n_1, n) and Next* (n, n_k) for some program line n.

In procedure Second we have:

Next* (1, 2), Next* (1, 3), Next* (4, 7), Next* (3, 4), Next* (4, 3), Next* (6, 5), etc.

Relationship Affects models data flows in a program. As indicated in the model of Figure 10, relationship Affects is defined only among assignment statements and involves a variable (shown as an attribute of relationship Affects). Informally, the relationship Affects (a1, a2) holds, if the value of v as computed at a1 may be used at a2.

Let a1 and a2 be two assignment statements such that a1 modifies the value of variable v and a2 uses the value of variable v. **Affects** (a1, a2) holds if a1 and a2 are in the same procedure, and there is a control flow path from a1 to a2 (i.e., Next* (a1, a2)) such that v is not modified (as defined by the Modifies relationship) in any assignment or procedure call statement on that path (excluding a1 and a2).

In procedure Second we have:

Affects (1, 4), Affects (1, 8), Affects (1, 10), Affects (1, 12)
Affects (2, 6), Affects (2, 10),
Affects (4, 8), Affects (4, 10), Affects (4, 12), and
Affects (9, 10)

But Affects (9, 12) does not hold as the value of z is modified by assignment 10 that is on any control flow path between assignments 9 and 12.
Suppose you have the following code:

```
1. x=a;

2. call p;

3. v=x;
```

If procedure p modifies variable x, that is Modifies (p, “x”) holds, then assignment Affects (1, 3) DOES NOT HOLD, as procedure call ‘kills’ the value of x as assigned in statement 1. If procedure p does not modify variable x; then Affects (1, 3) HOLDS.

In the program below, Affects (1, 5) and Affects (2, 5) do not hold as both x and y are modified in procedure q. Even though modification of variable y in procedure q is only conditional, we take it as if variable y was always modified when procedure q is called (which is in sync with our definition of Modifies for procedures). We make this assumption to simplify analysis for basic SPA.

We suggest a better solution (for bonus points) that will relax this assumption yielding more accurate results for Affects. This will require inter-procedural analysis of control and data flows in SIMPLE programs.

Affects (3, 5) holds as variable z is not modified in procedure q.

```
procedure p {

1.      x = 1;
2.      y = 2;
3.      z = y;
4.      call q;
5.      z = x + y + z;          }

procedure q {
6.      x = 5;
7.      if z then {
8.              t = x + 1; }
        else {
9.              y = z + x; } }
```

At the same time, we see that the value assigned to z in statement 9 *indirectly* affects the use of z in statement 12. Transitive closure ‘Affects*’ caters for this.

Let a1 and a2 be two assignment statements. **Affects*** (a1, a2) holds if a1 and a2 are in the same procedure, and a1 has either direct or indirect impact on a2, i.e.:

Affects* (a1, a2) if

Affects (a1, a2) or

Affects (a1, a) and Affects* (a, a2) for some assignment statement a.

Consider the following program fragment as an illustration of the basic principle:

1. x=a;

2. v=x;

3. z=v;

modification of x in statement 1 affects variable v in statement 2 and modification of v in statement 2 affects use of variable v in statement 3. So we have: Affects (1,2), Affects (2, 3) and Affects*(1, 3).

In procedure Second, we have: Affects*(1, 4), Affects*(1, 10), Affects*(1, 11), Affects*(1, 12), etc.

To compute relationships Affects and Affects*, you will need to have a CFG and also relationship Modifies for all the procedures.

6.4 Summary of Program Design Models

For your reference, here is a summary of program design entities, attributes and relationships for SIMPLE defined in program design models. When writing program queries, we can refer ONLY to entities, attributes and relationships listed below.

Program design entities:

program, procedure
stmtLst, stmt, assign, call, while, if
plus, minus, times
variable, constant
prog_line

Attributes and attribute value types:

procedure.procName, variable.varName : NAME
constant.value : INTEGER
stmt.stmt# : INTEGER (numbers assigned to statements for the purpose of reference)
call.procName : NAME

Program design entity relationships:

Modifies (procedure, variable)
Modifies (stmt, variable)
Uses (procedure, variable)
Uses (stmt, variable)
Calls (procedure 1, procedure 2)
Calls* (procedure 1, procedure 2)
Parent (stmt 1, stmt 2)
Parent* (stmt 1, stmt 2)
Follows (stmt 1, stmt 2)
Follows* (stmt 1, stmt 2)
Next (prog_line 1, prog_line 2)
Next* (prog_line 1, prog_line 2)
Affects (assign 1, assign2)
Affects* (assign 1, assign2)

7 Querying Programs with *PQL*

In this section, we define *PQL* by examples.

7.1 General Rules for Writing Program Queries in *PQL*

PQL queries are expressed in terms of program design models described in the previous section and summarized in "Summary of Program Design Models". So in queries you will see references to design entities (such as procedure, variable, assign, etc.), attributes (such procedure.procName or variable.varName), entity relationships (such as Calls (procedure, procedure)) and syntactic patterns (such as assign (variable, expr)). Evaluation of a query yields a list of program elements that match a query. Program elements are specific instances of design entities, for example, procedure named "Second", statement number 35 or variable named "x".

In a query, after keyword **Select**, you list query results, that is program design entities you are interested to find and **Select** from the program or the keyword BOOLEAN. You can further constrain the results by writing (optional) conditions that the results should satisfy. These conditions will include **with**, **such that** and **pattern** clauses. In *PQL*, all the keywords are in bold font.

Here are examples of queries without conditions:

procedure p;

Select p

Meaning: this query returns as a result all the procedures in a program. You could then display the result, for example, as a list of procedure names along with statement numbers. You could also display procedures along with source code (i.e., the whole program) if you wanted to. The issue of displaying query results is handled separately from the *PQL* by the SPA components *Query Result Projector* and *User Interface* (Figure 1).

procedure p;

Select p.procName

Meaning: returns as a result names of all the procedures in a program.

procedure p; assign a; variable v;

Select <a.stmt#, p.procName, v.varName>

Meaning: query result may be a tuple of program items. Here the result will contain all possible combinations of all assignment statement numbers, procedure names and variable names in a program.

The above query examples did not include any conditions. However, in most situations, we wish to select only specific program elements, for example, you may want to select only those statements that modify certain variable.

You specify properties of program elements to be selected in conditions that follow **Select**. Conditions are expressed in terms of:

a) entity attribute values and constants (**with** clause),

b) participation of entities in relationships (**such that** clause),

c) syntactic patterns (**pattern** clause).

All clauses are optional. Clauses **such that, with** and **pattern** may occur many times in the same query.

There is an implicit **and** operator between clauses – that means a query result must satisfy the conditions specified in ALL THE CLAUSES.

Declarations introduce synonyms that refer to design entities. Synonyms can be used in the remaining part of the query to refer to a corresponding entity. So you can write:

stmt s;

Select s **such that** Modifies (s, “x”) – here we introduce a synonym ‘s’ for statement and constrain the result with a condition. As you can guess, in this query, you **Select** all the statements that modify variable “x”.

A typical query has the following format:

Select … such that … with … pattern …

The following are **general rules for computing query results**:

A query reports any result for which THERE EXISTS a combination of synonym instances satisfying all the conditions specified in such that, with and pattern clauses. The existential quantifier is always implicit in *PQL* queries.

The result of **Select** BOOLEAN is TRUE if there exists a combination of synonym instances satisfying all the conditions specified in the query; otherwise, FALSE.

In the following sections, we shall introduce queries by examples, referring to program design models explained before.

7.2 Allowable Arguments of Relationships in Program Queries

Arguments of relationships are program design entities such as procedure, statement (stmt), while, etc. Naturally, synonyms of suitable design entities can appear as relationship arguments in queries, for example, we may have Calls (p, q), where p and q are synonyms of program design entity ‘procedure’.

In addition, we adopt conventions that will allow us to write certain queries in a simpler way. An argument in a relationship can be as follows:

1) A synonym of a valid design entity (valid according to the defection of a given relationship)
2) A placeholder ‘_’ (that is a free, unconstraint argument) provided it does not lead to ambiguity. Therefore, Modifies (_, “x”’) and Uses (_, “x”) are not allowed, as here it is not clear if the ‘_’ refers to a statement or procedure.
3) A character string in quotes, e.g., “xyz” can be used as an argument whenever an instance of the design entity can be identified by that string. When a relationship argument is program, procedure, or variable then string is interpreted as its name (should be NAME in quotes). When relationship argument is constant then string is interpreted as its value (the string should be INTEGER in quotes, e.g., “2”).
4) An integer can be used as an argument whenever an instance of the design entity can be identified by integer. In relationship Next and Next*, integer arguments mean program line numbers, and in Modifies, Uses, Parent, Parent*, Follows, Follows*, Affects, and Affects* an integer argument means statement number.
5) A synonym of prog_line can be used for arguments of type stmt. Then the prog_line is interpreted as a statement number.
6) Synonyms of stmt, assign, call, while and if can appear in place of program lines in Next and Next* relationship. Statement numbers are then interpreted as program lines.

prog_line n, n2, n3;

Select n such that Next* (n, n2) and Parent* (100, n3) and Modifies (n3, “x”) and Affects (n2, 100) and Follows* (92, n3)

In Parent*, Modifies and Follows* n3 is interpreted as a statement number; in Affects n2 is interpreted as a statement number (it must be assignment).

7.3 Query Examples: Calls, Modifies, Uses

Q1. Which procedures call at least one procedure?

procedure p, q;
Select p **such that** Calls (p, q)
Explanation: declaration introduces two variables "p" and "q" that can be used in a query to mean "procedure". Keywords are in bold. This query means exactly: "**Select** all the procedures p **such that there exists** a procedure q that satisfies Calls (p, q)".
IMPORTANT: note that the existential quantifier is always implicit in *PQL* queries. That means, you select any result for which THERE EXISTS a combination of synonym instances satisfying the conditions specified in such that, with and pattern clauses.
Example: in program First, the answer is: First, Second
A better way to write this query is:
procedure p;
Select p **such that** Calls (p, _)
Explanation: This query is the same as the one above. Underscore '_' is a placeholder for an unconstrained design entity (procedure in this case). Symbol '_' can be only used when the context uniquely implies the type of the argument denoted by '_'. Here, we infer from the program design model that '_' stands for the design entity "procedure".

Q2. Which procedures are called by at least one other procedure?

procedure q;
Select q **such that** Calls (_, q)

Q3. Find all pairs of procedures p and q such that p calls q.

Select <p,q > **such that** Calls (p, q)
Example: in program First, the answer is: <First, Second>, <Second, Third>

Q4. Find procedure named "Second"

procedure p;
Select p **with** p.procName = "Second"
Explanation: Dot '.' notation means a reference to the attribute value of an entity (procedure name in this case).

Q5. Which procedures are called from procedure "Second"?

procedure p, q;

Select q **such that** Calls (p, q) **with** p.procName = "Second"

You can also write the above query in short form, referring to procedure p in Calls via its name:

Select q **such that** Calls ("Second", q)

Here, we infer from the program design model that the first argument of relationship Calls is design entity "procedure". By convention, "Second" refers to procedure name. We shall use short form whenever it does not lead to misunderstanding.

Q6. Find procedures that call "Second" and modify the variable named "x"

procedure p;

Select p **such that** Calls (p, "Second") **and** Modifies (p, "x")

Explanation: The **and** operator can be used in relationship conditions under **such that** clause and in equations under **with** clause. We infer from the program design model that the second argument of relationship Calls is design entity "procedure" and the second argument of relationship Modifies is design entity “variable”. By convention, “Second” refers to procedure name and “x” refers to the variable name.

7.4 Query Examples: Querying Control and Data Flow Information

Using the same rules, we can write queries about control and data flow relations.

Q7. Is there a control path from program line 20 to program line 620?

Select BOOLEAN **such that** Next* (20, 620)

Explanation: The result of **Select** BOOLEAN is TRUE if there exists a combination of synonym instances satisfying all the conditions specified in the query; otherwise, FALSE.

Q8. Find all the program lines that can be executed between program line 20 and program line 620.

prog_line n;

Select n **such that** Next* (20, n) **and** Next* (n, 620)

Q9. Is there a control path in the CFG from program line 20 to program line 620 that passes through program line 40?

Select BOOLEAN **such that** Next* (20, 40) **and** Next* (40, 620)

Q10. Which assignment statements are directly affected by variable “x” assigned value at program line 20?

assign a;

Select a **such that** Affects (20, a)

Explanation: We can refer to program statements via their respective program line numbers.

Q11. Which assignments directly affect a value assigned to a variable in the assignment statement at program line 120?

assign a;

Select a **such that** Affects (a, 120)

Q12. Which statements contain a statement (at stmt#=”n”) that can be executed after line 16?

prog_line n; stmt s;

Select s **such that** Next* (16, n) **and** Parent* (s, n)

Q13. Which assignments affect assignment (at stmt#=”n”) that can be executed after line 13?

prog_line n; assign a;

Select a **such that** Affects* (a, n) **and** Next* (13, n)

Q14. Find all statements whose statement number is equal to some constant.
stmt s; constant c;

Select s **with** s.stmt# = c.value

Q15. Find procedures whose name is the same as the name of some variable.
variable v; procedure p;

Select p **with** p.procName = v.varName

Q16. Find statements that follow 10:
prog_line n; stmt s;

Select s **such that** Follows* (n, s) **with** n = 10

7.5 Query Examples: Finding Syntactic Patterns

Syntactic pattern matching is defined based on abstract syntax of *SIMPLE*.

Q17. Find all while statements.

while w;

Select w

Q18. Find all while statements directly nested in some if statement.

while w;

if if;

Select w **such that** Parent (if, w)

Q19. Find three while loops nested one in another.
while w1, w2, w3;

Select <w1, w2, w3> **such that** Parent* (w1, w2) **and** Parent* (w2, w3)

Explanation: notice that the query returns all the instances of three nested while loops in a program, not just the first one that is found.

Q20. Find all assignments with variable "x" at the left-hand side located in some while loop, and that can be reached (in terms of control flow) from program line 60

assign a; while w;

Select a **such that** Parent* (w, a) **and** Next* (60, n) **pattern** a ("x", _) such that a.stmt# = n

Explanation: in a ("x", _), we refer to the variable in the left-hand side of the assignment via its name. Patterns are specified using relational notation so they look the same as conditions in a **such that** clause. Think about a node in the AST as a relationship among its children. So assignment is written as assign (variable, expr) and while loop is written as "while (variable, _)". Patterns are specified in **pattern** clause. Conditions that follow pattern specification can further constrain patterns to be matched.

Remark: The two queries below yield the same result for *SIMPLE* programs. Notice also that this might not be the case in other languages. Why?

assign a;

Select a **pattern** a ("x", _)

Select a **such that** Modifies (a, "x")

Q21. Find all while statements with "x" as a control variable

while w;

Select w **pattern** w ("x", _)

Explanation: We use a place holder underscore '_' as statements in the while loop body (stmtLst) are not constrained in the query (they are irrelevant to the query).

Q22. Find assignment statements where variable x appears on the left-hand side.
assign a;

Select a **pattern** a ("x", _)

Q23. Find assignments with expression x*y+z on the right hand side
assign a;

Select a **pattern** a (_, "x*y+z")

Explanation: To find matching assignments, you draw an AST for the expression x * y + z. Any assignment whose right-hand side expression matches exactly AST for the expression x * y + z, matches the above pattern. For example, a = x * y + z matches the above pattern, but a = x * y + z + v does not match the pattern.

Q24. Find assignments in which x * y + z is a sub-expression of the expression on the right-hand side.
assign a;

Select a **pattern** a (_, _"x *y + z"_)

Explanation: underscores on both sides of the x * y + z indicate that x * y + z may be part of a larger expression. Again, pattern matching is done on AST. For example, a = x * y + z + v matches the above pattern, but a = w + x * y + z + v does not match the pattern.

Q25. Find assignments with expression (x + y) * z on the right-hand side.
assign a;

Select a **pattern** a (_, "(x + y) * z")

Explanation: Pattern specification may include brackets.

Q7. Find all assignments to variable "x" such that value of "x" is subsequently reassigned recursively in an assignment statement that is nested inside two loops.
assign a1, a2; while w1, w2;

Select a1 **pattern** a1 ("x", _) **and** a2 ("x",_"x"_) **such that** Affects (a1, a2) **and** Parent* (w2, a2) **and** Parent* (w1, w2)

7.6 Summary of Rules for Syntactic Patterns

1. A sub-expression must be must be a well-formed expression in *SIMPLE* (see grammar of *SIMPLE*). For example, "- x + y","x * y + (z", "x = 2; y = 3" is not a well-formed sub-expressions.
2. There can be only one sub-expression specification in a given pattern. For example, pattern a (_,"x"_"y") is not valid.
3. Pattern matching is defined based on abstract syntax of *SIMPLE*. It must take into account priorities and associativity of operators. Operator * has higher priority than + and -. Operators + and – have equal priority. Operators are left-associative, meaning the operations of the same priority are grouped from the left to the right.
4. Underscores on both sides of an expression in pattern specification (e.g., a ("v", _"x * y + z"_)) is matched by sub-expressions, as well as expressions.
5. Incorrect patters specifications include: a ("v", _"x * y + z") , a ("v", "x * y + z"_)
6. Please check 'patternCond' in *PQL* grammar at the end of the Handbook.

7.7 Comments on Query Semantics

Examples in this section are meant to further clarify interpretation of program queries in *PQL*.

7.7.1 Implicit existential quantifier in query conditions

The existential quantifier is always implicit in *PQL* queries. That means, you select any result for which THERE EXISTS a combination of synonym instances satisfying the conditions specified in such that, with and pattern clauses.

procedure p, q;
Select p **such that** Calls (p, q)
Answer: This query means exactly: **Select** all the procedures p **such that there exists** a procedure q that satisfies Calls (p, q). In program First of Figure 2, the result is procedures "First" and "Second".

7.7.2 Implicit ‘and’ operator between query clauses

procedure p, q; assign a;

Select <p, a> **such that** Calls (p, q) **with** p.procName = “Second” **pattern** a (“x”, _)

To qualify for the result, a procedure and assignment statement must satisfy all the conditions specified in the query.

Notice that a query may have any number of **such that**, **with** and **pattern** clauses that can be mixed in any order. There is an implicit **and** operator among all those clauses.

7.7.3 Use of free variables

A free variable is not constrained by any condition in the query. Underscore denotes a free variable. You can always replace underscore with a synonym. The following two queries are equivalent:

procedure p, q;

Select p **such that** Calls (p, _)

Select p **such that** Calls (p, q)

Use of underscore must not lead to ambiguities. For example, the following query should be rejected as incorrect as it is not clear if underscore refers to a statement or to a procedure:

Select BOOLEAN **such that** Modifies (_, “x”)

7.7.4 A note on meaningless queries

The Query Processor must be prepared to evaluate any query that is formally correct, even if some of the formally correct queries may have little use or make little sense. You could try to detect some classes of syntactically correct but meaningless queries during query validation and issue a warning message. But it is risky to reject and impossible to detect all such queries. In any case, the decision of which queries make sense and which ones do not make sense is somewhat arbitrary. Therefore, your Query Processor must be prepared to deal with all the queries that are syntactically correct.

Here are some example of “strange” queries:

procedure p, q; assign a, a1; while w; variable v; prog_line n1, n2;

(the above declarations apply to all the queries below.)

Select a

Answer: All the assignment statements in the program.

Select a **with** a.stmt# = 12

Answer: If statement at line 12 happens to be an assignment statement, then this assignment statement is selected; otherwise, the result is nil. Given numbering as in the program First of Figure 2, the result is assignment statement at line 12.

Select BOOLEAN **with** a.stmt# = 12

Answer: If statement at program line number 12 happens to be an assignment statement, then this the result is TRUE; otherwise, the result is FALSE. Given numbering as in the program First of Figure 2, the result is TRUE.

Select a1 **with** a.stmt# = 12

Answer: If statement at program line number 12 happens to be an assignment statement, then all the assignment statements in the program are selected; otherwise, the result is nil.

Select w **such that** Calls (“Second”, “Third”)

Answer: If procedure “Second” happens to call procedure “Third” in the program, then all the while statements are selected; otherwise, the result is nil. Given the program First of Figure 2, the result includes all while statements in the program.

Select <a, w> **such that** Calls (“Second”, “Third”)

Answer: If procedure “Second” happens to call procedure “Third” in the program, then a set of pairs including all the combinations of assignment and while statements in the program are selected; otherwise, the result is nil. Given the program “First” of Figure 2, the result is a set of pairs including all the combinations of assignment and while statements in the program

Select <p, q> **such that** Modifies (a, “y”)
Answer: This query means exactly: **Select** all the pairs <p, q> **such that there exists** an assignment a which modifies “y”. Even though p and q are not related to assignment a, the query is correct. If there is an assignment statement modifying “y”, the result is a set of pairs including all the combinations of procedures in a program.
Select BOOLEAN **such that** Calls (_, _)
Answer: If there is a procedure that calls some other procedure in the program, the result is TRUE; otherwise, the result is FALSE.
Select a **such that** Calls (_, _)
Answer: If there is a procedure that calls some other procedure in the program, the result includes all the assignment statements in the program; otherwise, the result is nil.

7.8 A General Format for *PQL* Queries

Please refer to Appendix A for a complete list of *PQL* grammar. The following is a general format of *PQL* queries:

select-cl : declaration* **Select** result-cl (with-cl | suchthat-cl | pattern-cl)*

Asterisk ‘*’ means repetition 0 or more times. Declarations introduce variables representing program design entities. Synonyms can be used in the remaining part of the query to mean a corresponding entity. The *result*-cl clause specifies a program view to be produced (i.e., tuples or a BOOLEAN value). In the *with*-cl clause, we constrain attribute values (e.g., procedure.procName=“Parse”). The *suchthat*-cl clause, specifies conditions in terms of relationship participation. The *pattern*-cl describes code patterns to be searched for.

Notice that a *PQL* query can contain any number of **such that, with** and **pattern** clauses and there is a default **and** between any two consecutive clauses. We can swap clauses without changing the meaning of the query.

All the clauses may appear more than one time in a query, in any order:

Select … **with** … **such that** … **with** … **with** … **pattern** … **such that** …

8 Required Program Quality Attributes

In addition to implementing SPA functionalities as described in assignment instructions, your solution must meet additional quality attributes. In this section, we describe the required quality attributes of your design, program and documentation. You must apply methods described in this Handbook in order to meet the required quality attributes. Further guidelines (particularly, regarding assignment and final report format) are given in assignments.

8.1 Quality of Architectural and Detailed Design Decisions

You make important decisions from architectural design to implementation. For each design problem, there are many design choices that have different pros and cons. Design decisions you make are interdependent and they should collectively contribute to meeting overall project goals. There is no “best solution”. Design is about evaluating trade-offs among various design decisions, and understanding the impacts of choices we make. What matters is not only the design decision itself, but also the process of arriving on design decisions, how you evaluate and argue about them, and their documentation.

8.2 Reliability of Your Programs and Testing

Your SPA should be well tested. Testing is an integral part of development. You do not defer testing until the end of the project. Each development iteration should result in a **production quality mini-system**. Each iteration should involve unit testing, integration testing and even some system testing. You should do integration testing after having released every significant piece of work. Why do integration testing often? It will save you lots of work. Integration testing shows if the major components in your system are in sync or not. Integration errors often arise in team project due to miscommunication among team members and making wrong assumptions based on imprecise documentation of component interfaces. Tackle these errors as early as possible. If they go unnoticed to later phases of implementation, it will cause you much time and pain to fix them.

Make a habit to incorporate unit testing into your everyday programming. Save test cases so that you can reuse them later when your program changes (regression testing). Use assertions to catch errors as close to the source of error as possible.

8.3 Flexibility of the SPA

You should design SPA for ease of making changes. In particular, it should be easy to modify SPA in the following ways:

1. Extend the source language *SIMPLE* with new language constructs.
2. Extend the PKB with new program design abstractions.
3. Modify data structures in which you store program design abstractions in the PKB.
4. Extend *PQL* with new query formats.
5. Refine Query Processor to perform optimizations during query evaluation.

To achieve flexibility, you must design SPA for change. The problem starts when a change you need to do has a global impact on the system and you have to revise lots of code in order to implement the change. So you will design SPA so that the impact of changes is localized to a small number of SPA components. You can limit the impact of changes by applying information hiding, careful design of interfaces among SPA components and with table-driven approach.

Design for change is in line with software development life cycle for the project. You will develop the SPA incrementally, in iterations rather than in one big-bang release. Three development iterations are defined in assignments 3-5. Each new iteration usually will involve some changes to the program base you had built in previous iterations. If you do not plan for change, moving through the iterations will cost you lots of extra work. While you cannot anticipate the impact of change 100%, to some extent you can. Sometimes you may need to re-factor your design to accommodate required extensions and to make your programs more flexible. This is a normal process involved in iterative development but of course, you want to minimize the amount of rework.

8.4 Reusability

It should be possible to reuse SPA components (after proper adaptations) in other similar systems, in particular, in SPAs for other source languages and in SPAs using different media for PKB (e.g., a relational database).

8.5 Scalability

Your SPA should work for input programs of 1000's lines of SIMPLE code. Make sure that your design decisions scale. In particular, be careful when pre-computing information for query evaluation. We require that relationships Next*, Affects and Affects* are computed "on demand" when evaluating queries, rather than pre-computed and stored in memory. The latter will not scale. Solutions that do not scale will affect project grade in a negative way.

8.6 Performance of Query Evaluation

Performance is a major problem in implementing descriptive notations for program queries such as *PQL*. Of course, query evaluation time depends predominantly on the size of the source program. However, design decisions considering the PKB and Query Processor also have much impact on efficiency of query evaluation.

The choice of data representation in the PKB affects the performance of query evaluation. If you design your data structures (for AST, CFG, etc.) taking into account how you will need to compute information during query evaluation, your Query Processor will run faster. Let us say you have stored only forward links among your CFG nodes. Evaluation of the query that requires traversing the CFG backwards will be very time consuming. The way you store information about variables modified/used will have much impact on the evaluation time of queries that involve relationship Affects().

Query Processor can optimize program queries for better performance. Refer to Section 16.8.3 in “Technical Tips” for further discussion of query evaluation and possible optimizations.

8.7 Quality of Program Documentation and Project Reports

Documentation you write must be understandable to other team members, not only to you. Make your documentation clear, precise, easy to follow and always keep it up to date with programs. Adopt documentation and programming standards so that all team members use the same conventions.

Proper documentation of architecture, component interfaces, use of assertions, documentation of test plans and test cases – all these are essential elements of the quality documentation. Quality of documentation contributes much to the quality of assignment and final project reports, based on which we shall grade your work.

--- Part II: Software Architecture and Design ---

9 Architecting SPA – Towards the Program Solution

Once basic SPA requirements have been understood, the time comes to think about the program solution that meets those requirements. Often, it is good to try out some implementation during requirement analysis. This early prototyping helps developers gain better understanding of requirements, foresee potential problems and get insights into the design of software architecture. Sometimes, a prototype (or certain parts of it) may be refined and included into the future system. For that reason, we asked you to do implementation exercises in assignments 1 and 2.

9.1 What Is a Software Architecture?

Software architecture provides a master plan for subsequent development phases and a blueprint for a software product itself. Software architecture is critical for coordinating project work and for ensuring that the final product meets its functional and quality requirements.

Many books have been written on software architecture, and you may come across many different interpretations of the term "software architecture".

In essence, a software architecture should play the following major roles:
Role 1. to tell us quickly how major functional and quality requirements of the SPA will be met,
Role 2. to explain, at conceptual level, how the SPA will work,
Role 3. to describe SPA components and their interfaces,
Role 4. to provide the basis for division of work among team members, that is, to tell who is doing what and what are the interfaces among team members, and
Role 5. to provide the basis for planning development iterations.

In great simplification, we can say: Architectural design is about decomposing a system into components (at various levels of abstraction) and precisely defining component interfaces. "At various levels of abstraction" means that we may start by identifying large granularity, subsystem level components (such as SPA front-end, PKB, query processing and user interface subsystems in Figure 11) and then continue decomposition of subsystems into lower level components as long as this is needed. For example, parser and design extractor are lower-level components in the SPA front-end subsystem.

9.2 SPA Architecture: Component View

Having good decomposition of a system and clear understanding and definition of component interfaces is a critical success factor for this project. Interfaces are contracts between components: the interface defines services a component promises to deliver to other components. Having decomposed a system and having defined interfaces, you can describe how your system works (Role 2) and how it meets requirements (Role 1). You can also let different team members work on different components in a fairly independent way. Component interfaces effectively become a communication channel and contractual interface between team members: As long as your team agrees on interfaces, team members can work on different system components. Interfaces ensure that, at the end, components developed by different team members can be successfully integrated. In that way, an architecture with explicit component interfaces makes it possible to divide work among team members (Role 4). Finally, based on architecture, you can decide what to do first and what to do next during development (Role 5).

The top level decomposition of an SPA (a refinement of Figure 1) is given in Figure 11. We have three subsystems, namely SPA Front-End, Program Knowledge Base (PKB), and Query Processor. We also have Query Result Projector and user interface components (they are too small to call them subsystems).

Arrows indicate information flows (not invocations!) among SPA subsystems and components.

SPA architecture of Figure 11 is based on common sense – the author of this Handbook can hardly imagine other top-level decomposition of SPA into subsystems/components. If you have ideas for other top-level decompositions – please do discuss those during consultations.

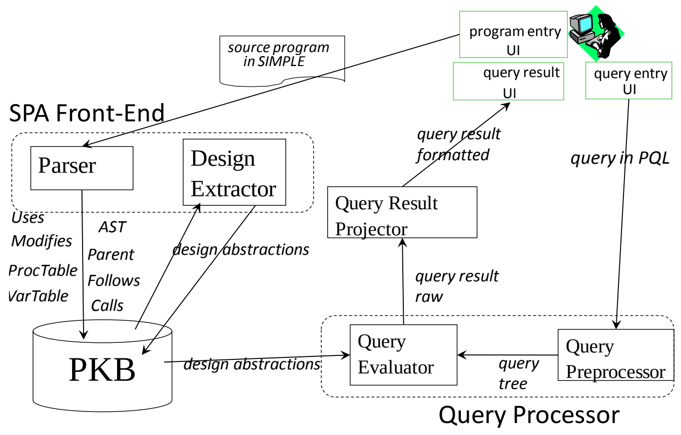


Figure 11. Top level decomposition of the Static Program Analyzer

9.2.1 An SPA Front-End subsystem

An SPA Front-End builds a PKB. As not all the design abstraction can be conveniently extracted during parsing, we decompose Front-End into *Parser* and *Design Extractor*: Parser builds AST and extracts design abstractions that can be easily computed during a single reading of the source program, and Design Extractor traverses AST to compute all the remaining design abstractions. Design abstractions that are easily computed by the Parser are listed next to the arrow from Parser to PKB.

Some solutions have been established in long history of building compilers and program analyzers, and we suggest that you employ them in the project. One such solution is to build symbol tables to store names program variables and procedures. Your SPA Parser can build variable table (VarTable) and procedure table (ProcTable).

Index	Variable Name	Extra info about a given variable	
1	x		
2	y		
3	z		
4	i		

With a VarTable, AST (and all other design abstractions stored in PKB) can refer to variables via their indexes in VarTable (integer number) rather than via variable names as character strings. Of course, all subsequent processing (during design extraction and query processing) also refers to variables via their indexes in VarTable. In case there is a need to keep extra information about program variables (suppose *SIMPLE* variables are typed?), VarTable is a place where you can store such information. ProcTable is analogical to VarTable.

Notice that AST and symbol tables contain complete information about a source program. Once an AST is created, there is no need to analyze program text anymore. Other program design abstractions can be more conveniently derived from the AST rather than by repeatedly parsing the program text.

9.2.2 Query Processor subsystem

As Query Processor must validate and evaluate queries, these two tasks can be allocated to components called Query Preprocessor and Query Evaluator, respectively. In addition to validating queries, a typical Query Preprocessor builds an internal representation for queries (referred to as query tree) that are used by Query Evaluator to evaluate queries.

9.3 A PKB Subsystem

During architectural design, your main task will be to further decompose the PKB subsystem and provide detailed definition of component interfaces.

PKB provides a storage (in terms of data structures) for program design abstractions such as AST, CFG, Calls, Modifies, Uses and others. Design abstractions such as AST, CFG, Modifies, Uses are stored in the data structures of the PKB. Design abstractions such as Follows*, Parent*, Next*, Affects and Affects* are not stored in an explicit form, but rather computed on demand during query evaluation.

The PKB will expose a *public* API (application program interface) to its clients (e.g., to the SPA front-end and the query processing subsystem), while hiding all the data structures used for storing program design abstractions in the PKB (*private* part). This important design decision is dictated by two considerations:

1. *Flexibility and reusability requirement*: the actual data structures to store program design abstractions are likely to change throughout development. Should this happen, you want to minimize the impact of change. PKB API will ensure that changes to data structures in the PKB will not affect the rest of the system.
2. *Division of work*: having agreed on the PKB API, work on SPA front-end, PKB and query processing subsystems can be done in parallel by different team members.

The PKB API plays a central role in the SPA architecture and in your project.

Each of the design abstractions, such as AST, CFG, Calls, Modifies, Uses, etc., will have its own *public* API exposing interface operations to its clients. Therefore, the PKB API will be a union of APIs for individual design abstractions. Implementation details of each design abstractions will be *private,* hidden from the clients. You can define APIs for individual design abstractions one by one, and at the end you will come up with the PKB API.

Your final PKB API will also include suitable interface operations to work with design abstractions "computed on demand" such as design abstraction Follows*, Parent*, Next*, Affects and Affects*.

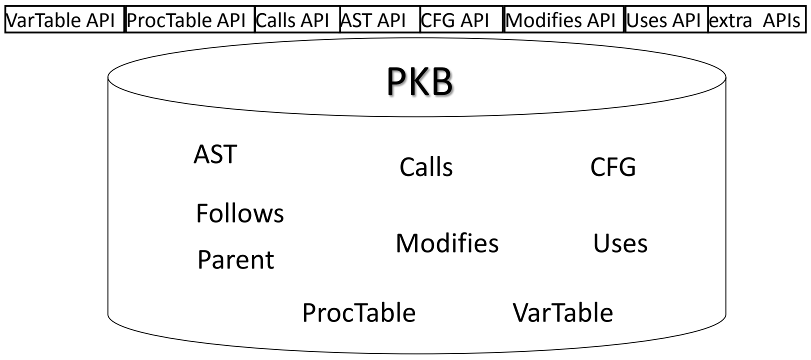


Figure 12. PKB API as a union of APIs for design abstractions

To define PKB API for a given design abstraction, you will have to understand how different clients (i.e., SPA components) are going to use PKB. PKB API should make it easy for clients to create design abstraction or access information about it, without knowing the design abstraction's data representation or any other implementation details. For the SPA front-end, the PKB API should make it easy to create an AST, CFG and other design abstractions such as Modifies, Uses, etc. For the query processing subsystem, the PKB API should make it easy to traverse the AST and CFG, and retrieve other information stored in the PKB that is required to validate and evaluate queries. You will have to analyze and understand the needs of different clients before you can design a good PKB API.

9.4 Abstract versus Concrete PKB API

PKB API describes how to work with design abstractions stored in the PKB. SPA components will communicate with PKB via PKB API. For example, PKB API will include operations to create AST nodes and link them together to form a tree structure (for the parser), and operations to traverse AST (for the Query Evaluator).

You will identify and specify PKB API at two levels: First at the architecture-level, abstract PKB API, and then at the program-level, concrete PKB API in C++, in SPA code.

Abstract PKB API will be described symbolically, using informal notation, while *program-level, concrete PKB API* will be written in C++, and will consist of public interfaces of classes implementing various design abstractions in PKB. You will design abstract PKB API first. Then, you will derive concrete PKB API from the abstract one. You will need to update abstract PKB API and keep it in sync with concrete PKB API throughout the project.

9.4.1 Benefits of abstract PKB API

There are many reasons why in bigger projects engineers choose to work at the level of abstract PKB API before writing program-level PKB API in C++:

1. It is always easier to understand essentials of a problem at hand if we are not overwhelmed with all the details. Abstract PKB API allows us to analyze, understand and document the essential behavior of design abstractions stored in PKB without considering how they will be implemented. We ask, for example "how will Design Extractor build a CFG?" and then identify interface operations for CFG API such as createNode(), setNextLink(FromNode, ToNode), etc. We need not know C++ to do that.
2. As abstract PKB API are free of implementation details and expressed in informal (or semi-formal) notation, they are much simpler, shorter and easier to understand than program-level, concrete PKB API.
3. You can implement the same abstract PKB API in many different ways, in many programming languages.

9.4.2 How to use abstract PKB API in the project

1. *Communication*: In team projects like yours, abstract PKB API specifications form a contract between team members. Team members communicate in terms of abstract PKB API. Based on abstract PKB API, you agree on how design abstractions will be used, clarify the meaning of various interface operations. You will have less miscommunications in team discussion.
2. *Division of work among team members*: Use abstract PKB API in discussion and assigning specific jobs to team members
3. *Guide to implementation*: When it comes to implementation, operations in abstract PKB API will become public methods (called member functions in C++) of PKB classes. You can refine them to add necessary implementation details, but the core concept of API remains the same. You will have less errors in a program.

To observe the above benefits, you need to keep your abstract PKB API in sync with your implementation. This does not mean that you retrofit implementation details into abstract PKB API. Abstract PKB API will fail to play its role if you did that. But it should be easy to trace from abstract PKB API to corresponding implementation classes (concrete PKB API). Using the same naming conventions in abstract PKB API and corresponding implementation classes (the same names in abstract operations/arguments and member functions), and using the same symbolic names for data types solves this problem. 'typedef' can easily assign specific C++ data types to symbolic names. Learn to communicate with your team mates in terms of abstract PKB API rather than in terms of internal implementation details.

9.5 The Process of Designing Abstract APIs

Here we illustrate general rules of how to design abstract APIs, illustrating with examples from PKB. The following three steps lead to understanding API requirements and API specifications:

1. Discover partial APIs
2. Consolidate and refine APIs
3. Formalize API specifications

9.5.1 Discover partial APIs

How do we figure out what interface operations a given design abstraction should have?

To define an API for a given design abstraction, you will have to understand how different clients (i.e., SPA components) are going to use that design abstraction. Sample interactions among SPA components and design abstractions are shown in Figure 13.

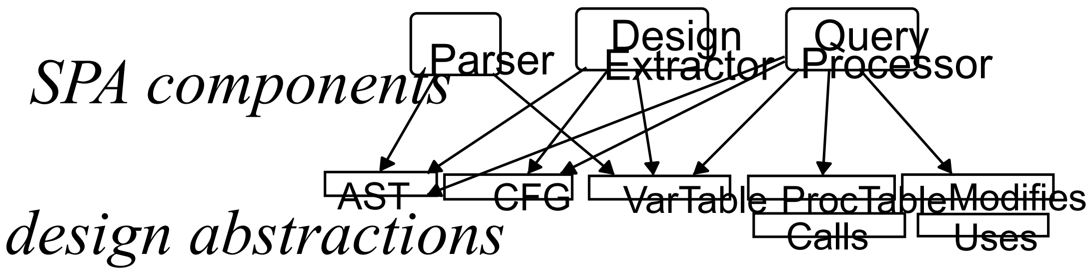


Figure 13. SPA components interacting with design abstractions

Use sequence diagrams to depict all the interactions. Based on that, you can work out a plan to systematically discover API for each interaction such as Parser → AST, Design Extractor → AST or Query Processor → AST. Then, you can consolidate partial AST APIs into a complete AST API. You can work with other design abstractions in a similar way.

A good strategy is to work in pairs, with one student taking the role of a design abstraction stored in PKB (such as VarTable, AST, CFG, or Calls), and the other student taking the role of a client (Parser or Query Evaluator) that creates the design abstraction or retrieves information from it.

For example, to discover AST API suitable for the parser, simulate parser execution on a small chunk of a *SIMPLE* program. Identify operations the parser needs to create AST nodes and links among them. These operations will form a part of the AST API. Then, simulate query evaluation and see what operations on AST you need to answer queries. In that way, incrementally, you will discover PKB API.

The following dialog illustrates a sample process of discovering an abstract AST API suitable for the parser:

Parser: I need to create an AST. Can you provide me with simple but flexible means to do that? I definitely do not want to be concerned with how you implement AST.

AST: That's great. I still consider a number of options of how to implement AST. So if you base your decisions on my implementation, you would have to wait. Also, if I decide to change my implementation later on, which is likely to happen we are bound to run into constant problems and lots of re-work. So I will give you an AST API – a set of interface operations to create AST. You will be able to work with AST using this AST API.

Parser: At the moment, you have not implemented AST and I have not implemented a parser. Shall we then just forget about implementation and come up with an abstract AST API that logically makes sense, based on common sense understanding of essential properties of AST?

AST: That is fine. We can agree on basic assumptions regarding AST API without being concerned with C++ (which btw I am still learning), doing analysis and initial validation of assumptions "on paper". We'll build a solid base for implementation. I will implement AST API and you will use it to build AST during parsing. So tell me – what do you need?

Parser: As I parse a SIMPLE source program, sometimes I need create an AST node for assignment statement or variable; at other times I want to designate a 'variable' node to be a child of an 'assign' node; or I want to designate a 'variable' node to be a child of 'assign' node or "+" node; or I want to designate a '+' node to be the right sibling of a 'var' node. The possibilities are endless, I am not in the position to enumerate all those specific cases; it is overwhelming. We definitely need a smart way to do it.

AST: Agree. So how about we start with an operation to create an AST node, createNode (). It will return to you a reference TNODE that uniquely identifies AST nodes. In that way you will be able to keep track of nodes you created. So I propose operation: TNODE createNode().

Parser: But when I create a node I also know which syntactic entity it represents – whether it is "var", "assign" or "constant"– and I would like to record this information at the node.

AST: I could give you operations such as createVar(), createAssign(), createConst(), etc. But then we'll have too many operations – it will not be clear or simple. In addition, we will have to change AST API anytime you change SIMPLE. So how about we add a parameter for that purpose: createNode (SYNT_TYPE syntType), so that you can provide syntactic type as you create a new node?

Parser: Sure, and also we can have operation setRoot (TNODE tNode) to designate 'tNode' as a root of AST.

AST: Ok. Later on when I implement this AST API, I will use typedef to assign proper C++ data type to TNODE. We will have nice traceability between abstract API and implementation, and also any change of the C++ data types will be easy to accommodate that way.

Parser: How about linking nodes to form a tree?

AST: We will have a couple of link types: child, sibling, Parent, and Follows. For convenience of AST traversal, we may need links up and down the AST, as well as in the left and right direction. How about I give you operation in which the first argument will denote the type of the link: CreateLink (LINK_TYPE link, TNODE fromNode, toNode)?

Parser: That will be fine. We reduce the number of interface operations without compromising convenience or readability. But let us agree and document the exact meaning of each type of link.

AST: Is there anything else that you need?

The dialog continues and at some point AST may want to talk to Design Extractor and Query Evaluator who need to traverse AST and fetch information from AST nodes.

Suppose the responsibilities are as follows:

1. Jane works on the SPA parser,
2. Mary works on the design of AST,
3. Suzan works on the design of CFG,
4. Tom works on the query pre-processor, and
5. John works on the Query Evaluator.

You will need to schedule the following meetings:

Team members participating in meeting	The purpose
Jane, Mary	Define the interface between parser and AST; what operations are needed to create an AST?
Jane, Suzan	Define the interface between design extractor and CFG; what operations are needed to create a CFG?

Tom, Mary, Suzan	Figure out what information is needed to validate query, to decide if this information is part of the PKB, to define suitable interface .
John, Mary, Suzan	Define the interface between Query Evaluator and PKB; what operations are needed to retrieve information from the AST, CFG, etc.

<u>Actions steps: how does Parser build AST?</u>

We illustrate the exact method for discovering APIs with an example of Jane and Mary working on AST API for Parser. Jane and Mary write a short SIMPLE code and AST to be generated by Parser. Based on that, they will identify Parser actions to create AST.

```
procedure Main {
      x = x+y+z;
      while i {
            y = x+i*2; } }
```

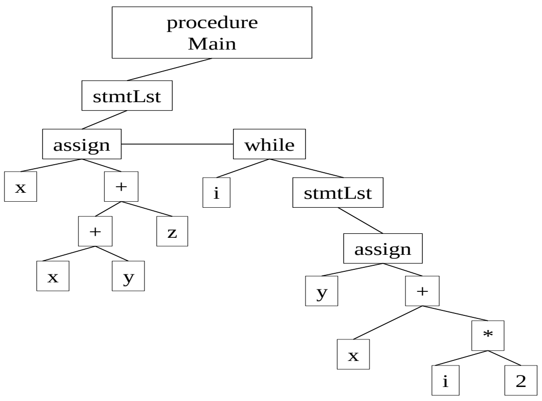


Figure 14. SIMPLE source and its AST

Starting analysis at the root of AST, Jane and Mary may come up with the following actions to build an AST that they can describe in plain English:

1. Create a node for procedure Main, return reference MainNode
2. Mark MainNode as the root of AST for Main
3. Create a node for stmtLst, return reference stmtLstNode
4. Link stmtLstNode as first child of MainNode
5. Create assign node, return assignNode
6. Link assignNode as first child of stmtLstNode
7. Create a node for variable x, return reference xNode
8. Link xNode as first child of assignNode
9. Create a node for “+”, return reference to plus2Node
10. Link plus2Node as right sibling of xNode (second child of assignNode)
11. Create a node for “+”, return reference to plus1Node
12. Link plus1Node as first child of plus2Node

Then, Mary and John could work in similar way to identify interactions between Design Extractor and AST. This would complete AST API with actions required to traverse AST.

9.5.2 Formalizing AST API

Here, you read through action steps for AST, and define operations that perform these actions. An operation will have a signature (operation name, arguments, and returned type) and brief description. You can introduce any symbolic names that you find useful in formalizing operation description:

TNode – reference that uniquely identifies AST node

Tnode CreateTNodeProc () – creates AST node for procedure and returns reference to it

SetAsRoot (TNode) – sets TNode as the root of AST

TnodeCreateTNodeStmtLst() – …

SetFirstChild (TNode P, C) – sets C as the first child of P

Continue until all action steps in AST description are represented by API operations.

9.5.3 Consolidating, refining and simplifying AST API

In this final phase, you consolidate APIs for Parser → AST and Design Extractor → AST. Unify notation used. Refine operation documentation according to general quality rules for API documentation discussed in Section 9.6.

Simply operations. At times, you can reduce the number of operations, without compromising their usefulness or readability. As an example, operations Tnode CreateTNodeProc() or TnodeCreateTNodeStmtLst(), could be replaced by one generic operation: CreateTNode (DE), where DE stands for a design entity: procedure, stmtLst, assign, variable, etc.

Identify other situations where you can avoid explosion of the number of operations.

To complete API documentation, you need look APIs across different design abstractions, and unify documentation style.

9.6 How to Document Abstract PKB API

It is important to document all the PKB APIs is a standardized way. Here is a suggested format:

Design Abstraction name {

Overview: explain in plain English the rationale and responsibility of a design abstraction

Public interface: here you will list interface operations documented as follows

Operation header: returned-value operation-name (list of parameters with names and types)

**Parameters (optional):* unless it is not clear from the header, describe parameters

Description: here you describe the effects of the operation (what the operation does) in terms of parameters, returned value and whatever else you need to explain the meaning of the operation; it is most important that you describe both normal and abnormal behavior (handled by assertions and exceptions)

}

When you describe your abstract PKB API, pay attention to the following recommendations. Refer to the list below often:

1. *Overview* clause: Should be brief, but for more complex design abstractions, such as AST, it is useful to provide some sketches and/or examples. Any new terms or names used throughout operations of a given design abstraction should be also described in the *Overview*.
2. There is no need to describe obvious parameters under the *Parameters* clause.
3. Choice of interface operations for the design abstraction.
   a) Interface operations should be well chosen to provide an easy way to work with the design abstraction.
   b) The set of interface operations should be complete in the sense that SPA components that work with a given design abstraction (create it or access information from it) can do so by means of calling interface operations.
4. Adopt common standards and use them consistently throughout all PKB API specifications:
   a) Use symbolic type names (e.g., INDEX, PROC, AST_NODE, CFG_NODE, etc.) rather than specific C++ type names (e.g., int).

b) Adopt naming conventions (for operation names, arguments, etc.).

c) There should be uniformity in the description of design abstractions: similar situations should be described in similar way. For example, VarTable should be similar to ProcTable. Modifies for statements should be similar to Modifies for procedures. Modifies should be similar to Uses.

5. *Description* of individual interface operations.

a) Keep the set of interface operations simple.

b) Interface operation specifications should be readable, understandable to others.

c) Interface operations should be abstract. Use of symbolic type names greatly helps keep interface operations implementation-and programming-language-independent.

i) *implementation-independent*: should not make unnecessary assumptions about implementation details such as the actual data structures chosen for design abstraction implementation.

ii) *programming-language-independent*: avoid specific C++ types.

d) Give names to arguments of interface operations and use these names in the operation *Description*. This helps to describe operations in a brief and clear way.

e) *Description* of interface operations should be simple, concise but precise, unambiguous, and complete. In particular, interface operation description should specify normal and abnormal behavior of the operation.

An example of VarTable API:

VarTable *Overview*: VarTable is used to keep all the variables that appeared in the program	
Public Interface:	
	INDEX insertVar (**STRING** varName); *Description*: **If** 'varName' is not in the VarTable, inserts it into the VarTable and returns its index. **Otherwise**, returns its INDEX and the table remains unchanged. OR: If variable is in the table, returns -1 (special value) and the table remains unchanged.
	INTEGER getSize(); *Description*: Returns the total number of variables in VarTable
	STRING getVarName (**INDEX** ind); *Description*: Returns the name of a variable at VarTable [ind] If 'ind' is out of range: Throws: InvalidReferenceException
	INDEX getVarIndex (**STRING** varName); *Description*: **If** 'varName' is in VarTable, returns its index; **Otherwise**, return -1 (special value)

9.7 Summary of Steps in the Design of APIs

1) Identify architectural SPA components

2) Understand associations among design abstractions (UML)

3) Understand interactions among SPA components and design abstractions (UML)

4) Discover API for each design abstraction, e.g., AST:

   a) For each interaction of SPA components with AST, describe in plain English how each component works with AST

   b) Consolidate partial descriptions of AST into a complete description

5) Work out initial API documentation standards

6) Formalize API documentation for each of the design abstractions

7) Refine and simplify APIs

   a) Refine API documentation standards and unify across all APIs

8) Discover and document any extra PKB APIs

   Extras cannot be associated with any design abstraction API

9.8 Concrete PKB API (Class Interfaces in C++ Program)

Suppose you have identified and documented your abstract PKB API. Abstract PKB API operations will form the core of public methods (called member functions in C++) in classes implementing design abstractions in the PKB such as AST or CFG. Concrete API refine abstract PKB API specifications with all the necessary implementation details. You can also add more interface operations to classes.

Following these two guidelines will help you maintain full traceability between abstract PKB API and relevant classes:

1. Do not retrofit implementation details into abstract PKB API. Otherwise, abstract PKB API will fail to play its role.
2. Use the same naming conventions in abstract PKB API and corresponding implementation classes (the same names in abstract operations/arguments and member functions). Use the same symbolic names for data types in abstract PKB API and corresponding implementation classes. 'typedef' can easily assign specific C++ data type to symbolic names.

10 From Architecture to Detailed Design

Many good books have been written on software engineering principles and methods, some of which you studied. We refer you to those books for a comprehensive discussion of software engineering principles. In this course, we adopt a "problem-based learning" approach: we want you to appreciate the value of good engineering principles by applying them to solve SPA problems.

The good news is that you are already familiar with most of the principles and techniques we want you to use. But there is a difference between just being familiar with the concept and knowing how to apply it to your benefit to solve practical problems. In this course, you will learn how to apply good engineering practices to achieve specific project goals.

10.1 Preliminaries: Common Sense Software Engineering Practices

Rule 1: Understand a problem before you go for full-fledged implementation

Make it a habit to analyze and discuss problems and their solutions **BEFORE** you embark on full-scale implementation. For this, you need identify and describe essential elements of a problem in an abstract way (rather than in terms of implementation details). Working with PKB APIs will be a great opportunity to learn how to do that.

Always focus on the WHAT of a problem first (requirements) and think about the HOW next (detailed design and implementation). Describe, discuss and evaluate WHAT before you decide about HOW. Learn to express your thoughts and communicate with other project stakeholders (e.g., team mates, project managers, customers, etc.) at the conceptual level. Once you know you have got the concept right, you will come up with better designed, simpler and more stable code.

How do you express the WHAT of a problem? You will use two major tools: modeling notations such as UML and component interface specifications. Diagrams and component interfaces convey essential properties of a software system and serve well as a description and communication mechanism during analysis of concepts, design and implementation.

Does it mean that you should not attempt implementing before you have completely understood the problem? No! Experiments and partial implementations are essential to obtain a complete understanding of a problem. For example, prototyping an SPA in a very simplified form will help you a lot to understand SPA's essentials. You must be prepared, however, to change (or even scrap) the results of this preliminary implementation work.

We shall now briefly reiterate (we assume you know it!) on three guiding principles, namely separation of concerns, simplicity and standardization, and information hiding.

Rule 2: Separation of concerns

Try to deal with one problem at a time. We already discussed separation of WHAT from HOW. In architectural design, you split a system into components so that you can cope with each component in separation from other components. When you model PKB, you concentrate on essential properties (e.g., associations among design abstractions in the PKB), describe them in separation from details that are irrelevant at this stage (such as the choice of data structure for AST). You also describe component interfaces in separation from implementation details. Finally, most often (and as it is also the case in this project) you develop system incrementally rather than in one big-bang release. In each iteration, you choose to concentrate on specific set of SPA functions in separation from other functions that you will address in subsequent iterations.

You can separate concerns in both space and time, that is, in the software product and in the software process. You can separate concerns horizontally (through decomposition) and vertically (through levels of abstractions). Separation of concerns is indeed one of the fundamental principles in software engineering and in problem solving in general.

Rule 3: The virtues of simplicity and standardization

There are always many ways of solving a given problem, and sometimes there will be also the "best" solution. At first, it seems that we should always aim for such a "best" solution. However, if we do so we will see many different solution patterns across a program. This will increase program complexity. Uniformity (or standardization) of solutions across a program is very important as it makes a program easier to understand and change by many people who work on a given program. Many "best" but incompatible solutions spreading over a program may appear counter-productive at the end. Of course, it does not mean that you should never innovate – there are plenty of problems that require creative and unique solutions for a good cause. It all depends on the problem in hand, the importance and value of such solutions. In each case, you need be aware if your unique (maybe the best) solution compromises uniformity and, if it does, there must be good reasons to apply it.

Rule 4: System decomposition with information hiding

Unlike hardware products, software must be flexible, easy to change. Unfortunately, we cannot design software to be flexible in all possible ways. However, if we know up front how the software is likely to change, we can design software for flexibility with respect to those specific types of changes. Take a minute to review Section 8 "Required Program Quality Attributes" to recall flexibility requirements for the SPA.

To achieve flexibility, you apply information hiding as a guiding principle during system decomposition. The idea is to minimize the impact of change. Design decisions that are subject to change must be hidden behind the interface. For example, clients of a design abstraction (such as SPA front-end and query processing subsystem) should not directly access its data structure. Instead, they should use interface operations. For example, AST will expose interface operations to create tree nodes, traverse the AST, etc., while hiding the linked list data representation of an AST.

Identifying system components, properly defining component responsibilities, identifying component interfaces and precisely describing them is the essence of software design. The most experienced software engineers are responsible for this task in industrial projects.

10.2 Becoming a Great Architect: Making Right Design Decisions

Any important design decisions and assumptions are part of a software architecture. An essential quality of software developers, and architects in particular, is the ability to make right design decisions. If you are good at it, you will be respected by colleagues and superiors, an indispensable member of a team, and highly rewarded by your employer. If you are the best – you will be a Chief Architect in your company. It takes the right approach, lots of common sense, and experience to become a great architect.

In this course, we show you first steps towards being a great architect: We tell you about the right approach to making design decisions. It is compulsory that you apply this approach throughout the project, and provide evidence (during consultations and assignment/final reports) that you have done so. Read this chapter carefully.

Any design problem can be solved in many ways. You make design decisions to choose a design solution that is right for problem at hand, and in sync with other design decisions. You will make hundreds of interdependent design decisions in the project, and they all collectively should lead to a successful product (SPA).

You make design decisions at all stages of software development as you progress from system requirements to program solution. We group design decisions into the following two categories:

1. Architectural design decisions

Software architecture includes system decomposition into main functional components, communication among components, and description of important data repositories. Any other important design decisions that have global impact on your system structure, behavior or qualities (such as performance, reliability, maintainability and other qualities that matter in a given project) also come under the umbrella of architectural design decisions.

2. Detailed design and implementation decisions

Detailed design decisions have to do with the choice of data representation and algorithms to best meet system requirements, or to achieve required program qualities (e.g., performance). Detailed design decisions are constrained by architectural design decisions. Detailed design decisions can be articulated to some extent in a general, programming language-independent way. But often it is good to complement such descriptions with relevant implementation choices, expressed in terms of the underlying programming language (such as C++).

Big O notation is helpful in comparing relative complexity of design alternatives.

10.2.1 General approach to making design decisions

The thinking process behind making architectural and detailed design decisions is very much the same. Here is the general approach to making design decisions:

1. Describe the design problem under consideration.
2. Identify, write down and prioritize goals to be met by relevant design solutions .
3. Consider alternative design solutions to the design problem under consideration.
4. Evaluate strengths and weaknesses of each alternative design solution:
   a) Analyze how well the design solution satisfies identified goals.
   b) Analyze any other implications of a given design solution.
5. Analyze trade-offs among design alternatives to justify your choice of the design solution.
   a) In the above, address any important implications of a given design decision. Depending on the context, take into account the complexity of the implementation, performance, how well a given design solution blends with other design decisions, etc.
6. Justify your choice of the preferred design solution.

10.2.2 Architectural design decisions in SPA

Figure 11 depicts the high-level architecture of SPA: functional components (such as Parser, Design Extractor or Query Processing subsystem), PKB as a repository of program information to be shared by SPA components, and major control and data flows among SPA components (shown by arrows). The choices of operations in PKB API and their documentation are also an integral part of software architecture.

Architectural decisions in SPA refer to PKB API design. PKB API comprises APIs of all the design abstractions, and yet other operations that do not belong to any design abstraction (e.g., operations for Affects). Important design decisions concern the choice of interface operations, and PKB API documentation standards. Goals and required qualities to be met by design decisions related PKB API are discussed in detail in Sections 9.4 to 9.7 of the Handbook. Most important are completeness and ease of use of API operations for design abstraction. In documentation of individual operations, most important are readability, completeness of interface operation descriptions (which should include the description of normal and abnormal behavior), language-independence of operation descriptions, and consistent use of adopted standards.

10.2.3 Detailed design decisions in SPA

In SPA, detailed design decisions include the choice of data representations for trees (AST), graphs (CFG), Modifies/Uses relationships, and other design abstractions. Then, for algorithms, we have parsing, traversing AST or CFG, computing transitive relationships (Calls*, Next*, Affects or Affects*), and many others.

All your design decisions must collectively aim at satisfying SPA's functional and quality requirements in the best possible, practical way. SPA answers queries about programs written in SIMPLE. The following are top design goals for SPA: Query answers should be correct (with respect to the specifications provided in the Handbook), and SPA should be able to answer queries fast. Memory consumption during query evaluation is also a concern. SPA should scale to big source programs and complex queries.

Trying to satisfy the above competing design goals is a challenge. There is no "best solution" to satisfy them, but there are many possible solutions that will do that in a better or worse way. Many detailed design decisions you make will collectively determine the quality of your SPA solutions.

The choice of data representations affects the complexity of your algorithms. Both data representations and algorithms have to do with memory utilization, and overall complexity of the design solution (and the effort to implement it). The actual time to compute algorithms will also depend on the time to fetch required information from PKB. You need take into account the interplay among those factors to make right design decisions.

Examples of design problems in SPA

Here are examples of important design problems for which you need make and justify design decisions (it is a small, random sample, which is not meant to be exhaustive):

- Traversing AST: to answer queries involving patterns, Query Evaluator may need traverse AST many times.
- Traversing CFG: to answer queries with Next* (n1, n2), Query Evaluator will have to traverse all the control flow paths between n1 and n2 down and up. The choice of data representation for CFG will have impact on efficiency of CFG traversal.
- Computing Affects (or Affects*): to answer queries with Affect* (a1, a2), Query Evaluator will have to traverse CFG, checking sets of modified and used variables on the way. Design decisions regarding CFG and relationships Modifies/Uses will impact efficiency of Affects.
- Modifies/Uses: Query Evaluator will often check if a variable is modified/used in a given statement, procedure or CFG node. In other cases, Query Evaluator will have to find all the statements (or procedures) that modify/use a give variable. The choice of data representation for Modifies/Uses will have much impact on query evaluation performance.

10.2.4 Estimating algorithmic complexity: Big O notation

Big O represents relative complexity of algorithms. You learned about Big-O notation in earlier courses. You are required to use Big O in evaluating and justifying design decisions, whenever it is relevant. In particular, use Big O to represent complexity of computing design abstractions such as Follows, Follows*, Modifies, Calls*, Next*, Affects, and Affects* (among others).

Big O should be used in evaluating design decisions together with (not instead of) other factors that matter such as memory utilization, the time to fetch required information from the PKB (e.g., the time to check if a given statement modifies a given variable or not), complexity of implementation, etc.

Be sure that you use Big O clearly and correctly. For example, when you say that algorithmic complexity is O(n), say what 'n' stands for. Follow simple rules of how to correctly use Big O described in this simple summary of Big O essentials by William Shields.

10.2.5 Documenting architectural design decisions

This Handbook gives precise guidelines for documenting architectural design decisions related to PKB API. Please follow these guidelines.

Use UML sequence diagrams to document communication among SPA functional components and PKB. Your top-level sequence diagram should correspond to the SPA diagram of Figure 11. This diagram can be refined into more detailed sequence diagrams showing communication between SPA functional components and specific data abstractions in PKB.

10.2.6 Documenting detailed design decisions

Detailed design decisions are usually interdependent in the sense that you make a number of design decisions that together solve a design problem. Also, the same design solutions may be contributing to a number of design problems. Organize documentation of detailed design decisions to simplify understanding of this situation.

Scenarios below illustrate two different contexts in which you will be analyzing and documenting design decisions:

Scenario 1: Many design decisions contribute to solving complex design problem

In such cases, start with the description of the design problem at hand. Analyze interrelated design decisions that collectively solve the problem. For example, you may want to explain design decisions that led to efficient computation of Affects. Here, "efficient computation of Affects*" is your design problem at hand. There are a number of design decisions contributing to Affects such as traversal of CFG and checking which variables are modified/used at CFG nodes. You could explain how design decisions collectively contribute to efficient computation of Affects.

Discuss and compare alternative design solutions that you considered (follow the approach discussed in Section 10.2.1). Use Big O notation. Justify your design decisions.

Scenario 2: Design decision contributes to solving many design problems

In such cases, start with description of the design solution. Examples of such design solutions include data representation for Modifies (or Uses), and creating a bi-directional map between statement numbers and AST nodes. These design decisions contribute to solving many design problems such as efficient computation of Affects, Affects*, evaluation queries with patterns, etc.

Discuss and compare alternative design solutions that you considered (follow the approach discussed in Section 10.2.1). Use Big O notation. Explain which design problems the design solution contributes to. Justify your design decisions in view of the many design problems that it contributes to.

Hint: Pay much attention to the clarity of documentation of your design decisions. A good practice is to give each important design problem and design solution a unique name (id). Use this name consistently throughout descriptions of your design decisions. Use graphical views to depict interrelationships among design problems and solutions (i.e., to show which design solutions contribute to which problems, etc.). Clear documentation of design decisions will score you a higher grade.

10.3 Using UML in the Project

We recommend using UML class, sequence and activity diagrams.

You may want to draw diagrams for a number of reasons: Diagrams can enhance understanding of how SPA works, depict the essence of a general or detail design solution, help you plan testing, or in project management (e.g., in task allocation).

You can draw diagrams at different levels of abstraction, from modeling concepts (requirements), to design, to implementation. Use UML diagrams mainly to model concepts and design (see more recommendations at the end of this section).

This Handbook provides many examples of how class diagrams can be used to model concepts in SPA domain: design entities, relationships, AST, and design abstractions in PKB. Activity diagrams are quite natural to use. The rest of the discussion we dedicate to sequence diagrams.

10.3.1 Using UML sequence diagrams

A sequence diagram shows how a group of objects collaborate to achieve *important functions* of a given software system. In business systems, these *important functions* are identified by use cases. Therefore, we may have a sequence diagram per use case.

In SPA, queries play the role of use cases. Also, important functions such as computation of Affects may be considered as a use case.

Sequence diagrams can be used at all levels of abstraction, from system, to design and to implementation levels. High level sequence diagrams show system-level control flows among major subsystems (e.g., Parser, Design Extractor or Query Evaluator in SPA). Design level sequence diagrams can show control flows (and execution sequences) among components that are assigned more detailed responsibilities (e.g., Query Preprocessor or Query Result Projector).

In SPA, at high abstraction level, we can draw sequence diagrams to show how SPA functional components collaborate to answer queries. Such sequence diagrams have similar contents to high-level component view of SPA architecture shown in Figure 11.

Notice that in the sequence diagrams below, different icons are used for user interface elements (SPA UI), functional components (SPA Front-End and Query Processor), and data (PKB). Labels attached arrows are responsibilities of relevant components.

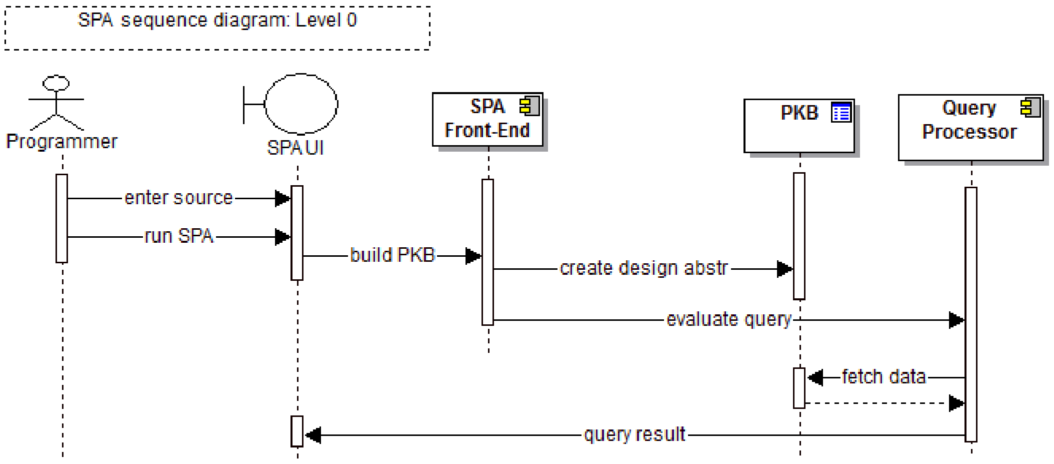


Figure 15. High-level view of SPA architecture

Model of Figure 15 is simple and intuitive. Next, we can start refining a model of Figure 15 with more details. We observe that a model of Figure 15 lacks proper coordination. Who calls the SPA Front-End? Is it right for the SPA Front-End to call the Query Processor? An important element of good design is to localize control.

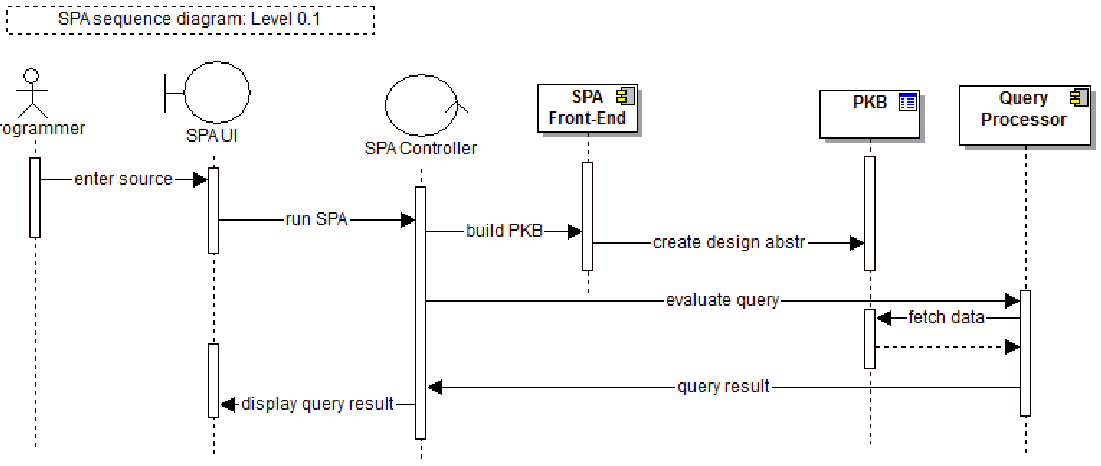


Figure 16. SPA architecture with Controller

In the refined model of Figure 16, we introduce an SPA Controller to play this role. In SPA implementation, main() program is likely to play the role of SPA Controller.

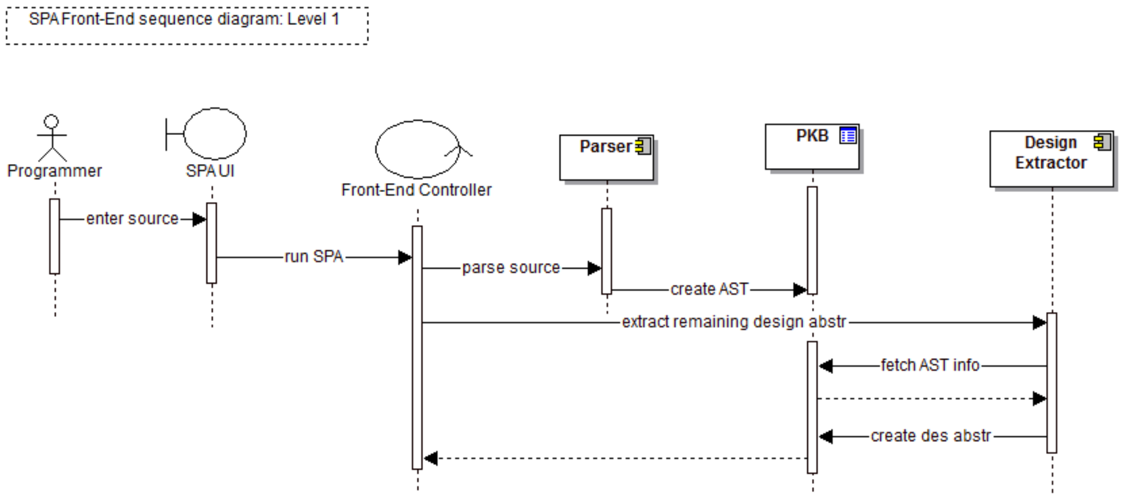


Figure 17. Front-End subsystem sequence diagram

In the refined model of Figure 17, we decompose the SPA Front-End into its two sub-components, namely Parser and Design Extractor (please compare this sequence diagram with SPA model of Figure 11).

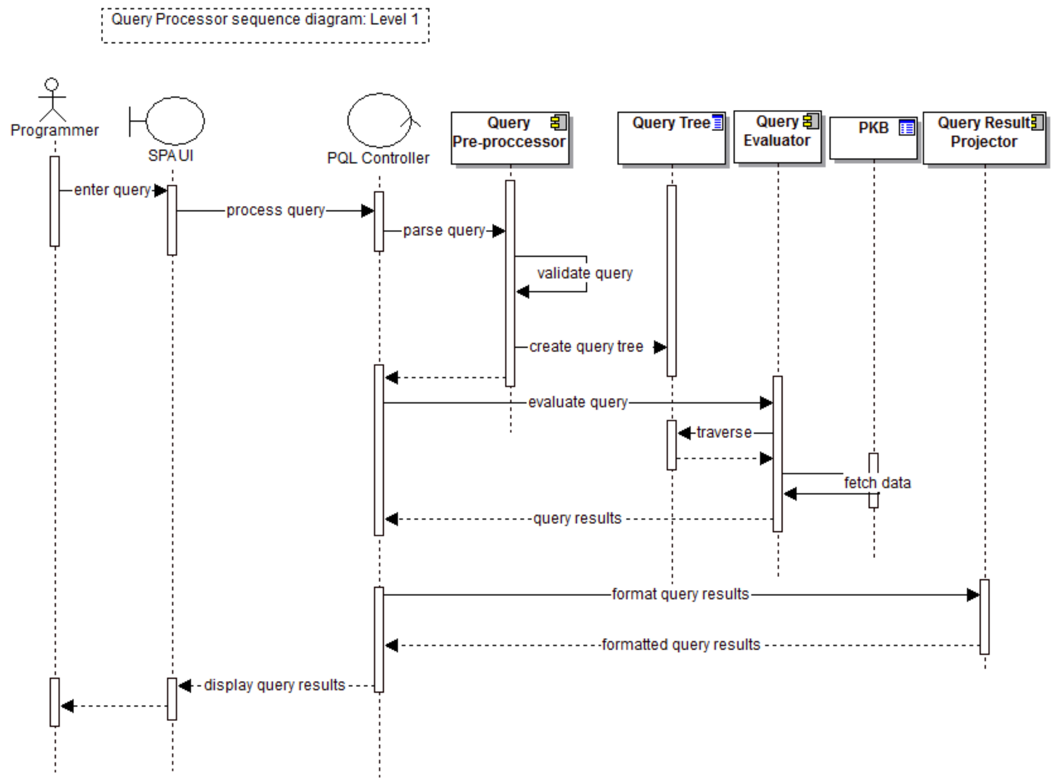


Figure 18. ***PQL*** **subsystem sequence diagram**

In the refined model of Figure 18, we decompose the *PQL* subsystem.

10.3.2 When to use sequence diagrams

1) Use sequence diagrams to document your design decisions. They will help you analyze alternative ways to design your SPA. All team members will share a clear picture of how SPA works.
2) Sequence diagrams will help you understand dependencies among SPA components and tasks assigned to different team members.
3) Use sequence diagrams in project planning: Annotate sequence diagram elements with tasks assigned to different team members.
4) Sequence diagrams will help you plan and monitor testing. During test planning, annotate sequence diagram elements with information about who and when will test different components and their responsibilities. Then, as you progress with testing, mark sequence diagrams that have been tested and the degree of their reliability.
5) Use models in any way you find them useful.

10.3.3 Summary of recommendations:

1) *Clear purpose*: Draw only diagrams that have a clear purpose. You should be able to explain in what ways a given diagram is useful. For example, diagrams can enhance understanding of the SPA problem, depict the essence of a general or detailed design solution, help you plan testing, or in project management (e.g., in task allocation).
2) *Use the diagrams in correct way*: Each UML model is meant to model certain aspects of system, but will not be useful if you use it to model other aspects. Use class diagrams to model concepts and structure, but not to model system behavior. Use sequence diagrams to show how functional components collaborate to achieve important functions. Use activity diagrams to show algorithmic details of complex computations. Always be sure which diagram to use for a task at hand.
3) *Cohesion*: Keep each diagram focused on one specific goal. Do not try to model everything in one diagram. For example, do not model structure (design abstractions in PKB) and behavior (Parser or Query Evaluator) in one class diagram. Not only will you misuse notation (see point 1) above), but

your diagram will also become unreadable, defeating the very purpose why you spent time to create it.

4) *Level of abstraction*: Do not overload diagrams with unnecessary details, as this will blur the picture. Remember that understanding is often the main reason why you create a diagram in the first place. Higher-level diagrams depicting concepts and design are generally more useful than low-level diagrams showing implementation. Implementation-level diagrams are often too big and too many; in addition, it is difficult to keep them in sync with evolving code.
5) *Traceability across diagrams*: Use naming or other conventions to ensure that it is easy to trace model elements across diagrams.
6) *Use of UML notation*: Use UML notation and organize diagrams to enhance readability. This is more important than rigorously following textbook examples. In case you do not use diagrams in standard way, always explain what you mean. Use stereotypes if you need extend UML conventions.

10.4 Design Patterns

Design patterns provide standardized solutions to design problems. You studied some typical design problems for which published design patterns exist in CS2103. If you find instances of those problems in your project, consider applying a suitable design pattern.

By applying a design pattern, you usually win more flexibility, but an overall program solution may be more complex to work with. Always evaluate carefully the trade-offs involved in terms of expected benefits and the cost of applying a design pattern. Apply a design pattern only if the benefits outweigh the cost.

10.5 Coding Practices

Write as simple code as it can be in view of functional and quality requirements that matter. Design is always a trade-off among many goals and considerations, some of which may be competing or even conflicting with others. Therefore, you will not be able to go for “the best possible” solution to each problem you tackle. From program understanding and maintenance viewpoint, uniform solutions are better than unique, optimized solutions. So be clear about main priorities when evaluation your design and implementation strategies.

There are good observations in Brooks’ *Mythical Man-Month* about wise use of creativity in big software projects. Companies favour uniform solutions over highly optimized but diverse and complex solutions. Standardized programs are easy to understand and share. The learning curve to join a team or to take over a task from others is very much reduced. Companies ask developers to follow standards (often formalized as patterns similar to GOF design patterns, but much more concrete). These standards are to ensure “conceptual integrity” of software developed by teams and across projects in a company. Standards protect certain project areas from free creativity, channeling developers’ creative powers to the areas that can truly benefit from novel solutions.

Set up naming and program formatting standards to make your code as readable as possible. Keep C++ naming conventions in sync with naming conventions for abstract APIs.

Some good examples are in references [1] and [2] below. Especially, check recommendations regarding the choice of meaningful names and the use of vertical alignment and white spaces to enhance code readability.

Go for simple detailed design and program solutions as long as there is no clear need to consider more complicated ones (e.g., for performance, extensibility or reuse reasons). As you go through iterations next term, things will get quite complicated anyway. Any extra and unjustified complexity of design solutions will also cause an extra pain. This danger will be quite real in INF2ATS, so try to avoid it.

As a rule of thumb, concise code is usually simpler and less error-prone than bulky code. Suppose you consider two solutions to the same problem: Solution 1 with 2 classes, 10 functions and 100 LOC (lines of code), and Solution 2 with 20 classes related by inheritance, 80 functions and 500 LOC. If Solution

1 is fairly natural and reasonably meets your requirements, then you need really strong arguments to go for Solution 2. Of course there may be yet other solutions in between Solutions 1 and 2 that can best meet your needs.

References:

[1] Robert Green, Henry Ledgard “Coding guidelines: finding the art in the science,” Communications of ACM, Dec. 2011, pp. 57-63 (download from NUS Digital Library)

[2] Ledgard, H. “Professional coding guidelines,” 2011 Unpublished report, University of Toledo; http://www.eng.utoledo.edu/eecs/faculty_web/hledgard/softe/upload/

--- Part III: Software Process ---

11 The Project Team

The design of an SPA involves the following major tasks:

1. Building a PKB:
   a) Design of the data structures to store the program design abstractions in the PKB.
   b) Design of the SPA front-end to parse a source program and build an AST.
   c) Design of the algorithms to traverse an AST and derive procedure call, control flow and other program design information, as described in the program design model.
2. Designing an application program interface (API) to PKB. PKB API implements all the operations that are needed for its clients, i.e., SPA front-end and query processing subsystem, in particular:
   a) To traverse ASTs up and down.
   b) To traverse the CFGs.
   c) To retrieve program elements related by means of relationships Follow, Follow*, Parent, Parent*, Next*, Affects, Affect*, as defined in the program design model.
3. Designing a query processing subsystem, in particular:
   a) Query Preprocessor.
   b) Query Evaluator.
4. Design of the user interface subsystem.

You will form teams of 4-6 students. Each team will be divided into two groups (2-3 students per group), called Group-PKB and Group-*PQL*, respectively. Roughly, Group-PKB will be responsible for task 1, Group-*PQL* will be responsible for tasks 3 and 4, and both groups will be equally responsible for task 2 – designing a PKB API.

The PKB API is critical to the success of the project. It must be well designed, taking into account data representation in PKB and the needs of the Query Evaluator. But you will not get it right at the first shot. So be prepared for refinements as project progresses. Members of the Group-PKB and Group-*PQL* will have to communicate a lot. To facilitate communication, it is essential that you document PKB API in a clear way and keep the document up to date with changes throughout the project.

12 An SDLC for the Project

To manage project complexity, you will develop the project incrementally, in iterations. Each new iteration will extend the program implemented in the previous iteration. The main purpose of iterative development is to tackle difficulties one by one. In this section, we discuss motivation and basic rules for iterative software development lifecycle (SDLC).

The SDLC for this project is a variant of the Unified Process. Read a brief introduction to the Unified Process in Chapter 2 ofM. Fowler's *UML Distilled, Second Edition,* Addison-Wesley, 2000. The SDLC and development iterations for your project take into account specific technical difficulties involved in the design of an SPA. Project phases include analysis, architectural design and construction (iterative development).

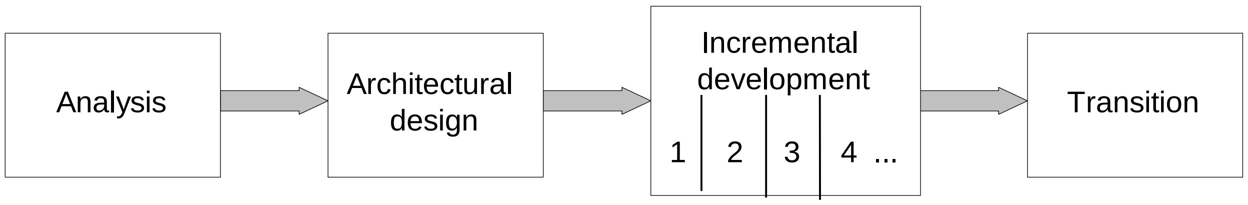


Figure 19. SDLC phases for the project

12.1 Analysis

During analysis, you get an understanding of:

- what product you are going to build, and
- how you are going to build it.

Analysis is risk-driven – you want to identify risks and address them early in the project. Risks can be related to requirements (e.g., unexpected changes, fuzziness), technical issues (e.g., complexity of algorithms), personnel (e.g., team member gets sick or lacks of expertise in some area) and, sometimes, politics (unlikely in your case). First, you need to analyze the problem description to be sure that you understand all the concepts. If you were implementing a business application, you would have to interview users, perhaps build screen prototypes to clarify requirements. You would create models of required system functions (such as use cases) and conceptual models (such as class diagrams). In this project, the problem has been described for you. But as the problem is complex, you will produce program design abstractions for sample programs by hand and play with them to get an understanding of the concepts. You will also evaluate by hand program queries for sample programs.

12.2 Architectural Design

During architectural design you will do the following:

1. Identify major components of the SPA and describe their responsibilities.
2. Describe how components will work together.
3. Document component interfaces.
4. Model associations among design abstractions as UML class diagram.
5. Describe in detail the meaning of associations.

The most important part of architectural design will be to define interfaces to major program design abstractions (such as ASTs, CFG and procedure call trees). These interfaces should hide physical representation of data and provide a convenient set of operations to retrieve information from the PKB during query evaluation. The union of interface operations for all program design abstractions stored in the PKB will be for the PKB API.

At the end of architectural design, you should be also clear about who will do what in a team and how you will communicate within the team during development.

12.3 Iterative Development

This is a construction phase in which you will build an SPA incrementally, in three iterations. At the end of each iteration, you will produce a fully tested and integrated program, implementing part of system functionality. Each iteration can be viewed as a mini-project, as it will involve some amount of analysis, design, implementation, integration and testing.

Iterative development is the key to successful development of complex and large systems. It is based on separation of concerns and "divide and conquer", principles that have been effectively used for years to tackle complex tasks in various engineering disciplines.

Often, a program you develop in one iteration will tackle only a simplified problem. Therefore, you will have to extend the program in the later iterations. Program flexibility and design for change with information hiding are essential in iterative development. This is not a limitation in this project as one of your goals is to design a flexible SPA that can be easily changed in many aspects. Planning for change from the beginning will only help you achieve this goal.

Obviously, the order and scope of iterations must be carefully defined to ensure smooth project development. Therefore, iterative development starts with planning. It is important to plan iterations so that:

1. problems attacked during each iteration are not overly complex, and
2. program solutions developed in earlier iterations can be easily reused and extended in later iterations.

Having read the description of the SPA, you may have an impression that it must a formidable task to implement it. In fact, in the scope as it was described – you are right, it is a large and difficult task, especially if you think about making it in one big-bang release. But the degree of difficulty depends on how you scope the problem. Suppose, for example, that you further simplify *SIMPLE* and assume that a program consists of a single procedure with a sequence of assignments such as x = 1 or x = y. In addition, you will only address queries of the following formats:

stmt s1, s2;

Select s1 **such that** Parent (s1, s2) **with** s2.stmt#= int?

Select s1 **such that** Follows (s1, s2) **with** s2.stmt#= int?

and answer them for different values of the argument 'int?' (i.e., statement numbers). Does it sound difficult? You will find out soon as we shall ask you to develop such an SPA prototype at the beginning of the project. If it looks too trivial to you – you can extend the scope of the prototype by addressing while or if control structures or by addressing queries in the following format:

assign a1, a2; variable v;

Select a1 **such that** Affects* (a1, a2) **with** a2.stmt# = ? **and** v.varName = ?

12.4 Planning Iterative Development

Iterations mark project stages at which programs reach certain level of maturity. While not fully stable, programs released at the end of each iteration are designed to facilitate changes that occur in subsequent iterations. Having analyzed a problem and designed a software architecture – how do you plan development iterations? You plan iterations based on:

- system requirements, i.e., functions to be implemented, and
- major internal structures that are required to implement a solution.

Let us compare the following two iteration strategies. Let us call the first strategy the depth-first strategy. In each iteration, you identify a problem (e.g., a system function you need to implement) and try to provide a complete program solution for that problem in one iteration (i.e., you go deep into the problem). In subsequent iterations, you add more functions one by one and you come up with the complete system at the end. For example, Group-PKB could start by developing a parser for *SIMPLE* and generating an AST for programs. At the same time, Group-*PQL* might develop a strategy for query validation and start designing a general mechanism for query evaluation.

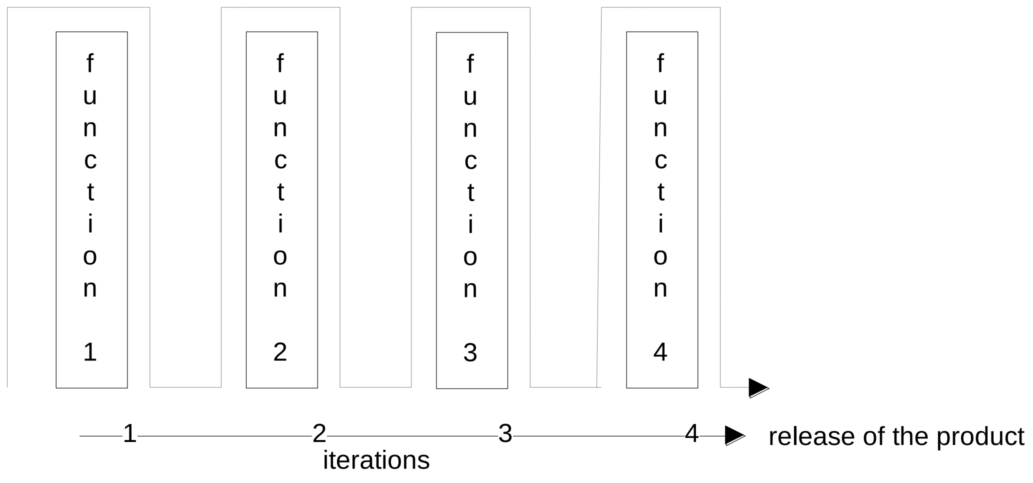


Figure 20. Depth-first iterative development

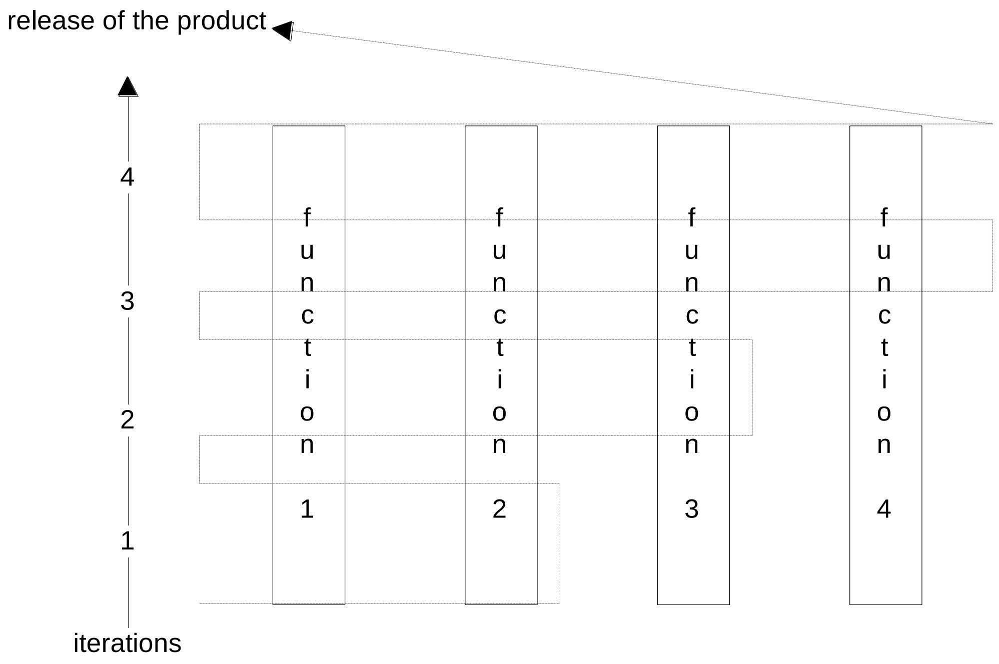


Figure 21. Breadth-first iterative development

Here is another idea for iterative development that we shall call the breadth-first strategy. Rather than digging deep into specific narrow problems, in each iteration we touch on and experiment with a number of related problems, but in a simplified form. We move broad and shallow through the problem and solution spaces rather than deep and narrow as before. Like in the example in the previous section, Group-PKB may start with a very small subset of *SIMPLE* and develop PKB containing a wide variety of program design abstractions for that subset. Group-*PQL* can start by implementing specific program queries, for the same subset of *SIMPLE* that Group-PKB covers. In subsequent iterations, Group-PKB would extend subset of *SIMPLE* and refine the data representation for program design abstractions in the PKB.

The following steps illustrate possible iterations in each strategy:

Depth-first iterations:
1. Develop parser for *SIMPLE*
2. Generate an AST
3. Develop AST traversal algorithms

4. Generate CFG
5. Parse queries, etc.

Breadth-first iterations:

1. Select small subset of *SIMPLE*
   Develop parser for subset, simplified AST, CFG
   Validate/evaluate some simple types of queries
2. Extend the subset of *SIMPLE*
   Extend AST and CFG, add more program design abstractions to PKB
   Address more types of queries
3. …

Which strategy is more appropriate for the SPA project? Sketch alternative plans for iterative development of SPA, and analyze the trade-offs among them.

--- Part IV: Techniques ---

13 Table-driven Technique – Query Validation Example

Table-driven technique is a simple yet powerful way to achieve flexibility. Consider the problem of query validation and the requirement that SPA should be flexible with respect to changes to a program design models. This means that we should able to easily add new design entities, change attributes of design entities and relationships.

First, let us describe what is involved in query validation. (This function is performed by the Query Preprocessor in Figure 9). As you enter query (or after you have entered query but before query evaluation), you need to check if the query is valid with respect to program design models for SIMPLE. That means, you need to check if all the references to entities, attributes and relationships in a program query agree with the program design model definition. In particular:

1. All the program design entities in declaration of synonyms should be defined in the program design model. In the case of *SIMPLE*, valid program design entities are procedure, variable, assign, while, etc.
2. All the relationships in query conditions should be syntactically correct. In particular, they should have required number of arguments and the type of arguments should agree to be the same as in the program design model. For example, in relationship Calls (p, q), p and q should be synonyms of the design entity procedure. Cluster (p, s) is not valid as we do not have relationship Cluster () in the program design model. Reference to Calls (p, q, v) is not valid either as Calls expects only two arguments of type procedure.
3. All the references to entity attributes should be valid. First of all, attributes should be defined in the program design model. For example, procedure.value is not a reference to a valid attribute. Equation p.procName=2 in **with** clause is not valid, as the type of attribute procName is a string not INTEGER.

How to design Query Preprocessor so that we can change program design models with minimum effort? You could hard-code program design models in the query validation algorithms. The code for query validation will contain switch statements such as:

```
switch ( relationship) {
        case Calls: expect two arguments; each argument should be either procedure synonym or ‘_’
        case Next: expect two arguments; each argument should be either statement number or
                synonym of statement, assign, if, while or ‘_’
        etc.
}
```

In case of changes to the program design model, you will have to find and modify affected parts of the code. Table-driven technique provides a better solution. Rather than hard-coding program design models into query validation algorithms, we define program design models in tables. For example:

1	procedure
2	stmtLst
3	Stmt
4	assign
5	etc.

Table 1. Entity table – EntTable

Relationship	# args	type of argument 1	type of argument 2
Calls	2	synonym (procedure), _, string	synonym (procedure), _, string
Calls*	2	synonym (procedure), _, string	synonym (procedure), _, string
Modifies	2	synonym (procedure | prog_line | stmt | assign …), string, INTEGER	synonym (variable) | string | _
other rels	..	…	

Table 2. Relationship table – RelTable

An entity attribute table can be defined in a similar way. All the elements of a program design model that you need during query validation are now described in the tables. A query preprocessor will refer to the tables to check whether all the references to entities, attributes and relationships in a query agree with the program design model definition. A query preprocessor will be more generic and much simpler now. Instead of switch statements, you will have the following code:

```
rel = getRelTabIndex (R)
if ( #args ≠ RelTab [rel, 2] ) Error()
if ( arg1 ∉RelTab [rel, 3] then Error()
if ( arg2 ∉RelTab [rel, 4] then Error()
etc.
```

where R is a relationship referenced in a query, #args is the number of arguments of R as it appears in the query, arg1 is the first argument of R, etc. Validation code checks if the actual references in a query agree with their respective definitions in the tables.

The advantage of the table-driven solution is that any changes to the program design models will affect only tables but not the code of the query pre-processor. Changing data in tables is much easier than changing procedural code.

14 Error Handling, Exceptions and Assertions in C++

Error handling is always a big issue in programming as we all make errors. Exceptions and assertions are means to handle errors in a systematic way. Assertions also play a role in documenting programs.

As there is an overlap between exceptions and assertions, some guidelines can be useful. This section provides such guidelines and pointers to further reading.

Notice that “guidelines” do not mean rules that have to be blindly followed. Sometimes the situation may call for a different solution than recommended by guidelines.

14.1 Exceptions

“There is an exception to every rule”. Exception mechanism caters for that. Each function has a typical (normal) behavior as well as cases of abnormal behavior. Abnormal behavior signifies that something went wrong and the possibility of an error. You can handle abnormal behavior by writing ‘if’ test and taking appropriate action when the condition is true. But it is better to use exceptions.

It is recommended, that each function defends itself from any possible erroneous calls. In particular, this means that a function should check pre-conditions and define actions to handle situation when pre-conditions are violated (it is called “defensive programming”). The following are advantages of making each function defend itself from possible errors:

1. Error handling code is defined in one place – in the function itself
2. Callers need not check pre-condition
3. Function becomes more reusable
4. Program becomes more reliable and easier to change/debug.

Exceptions are written to handle anticipated errors. When an error occurs, an exception code dedicated to the error can either print information that will help you identify the reason of an error in order to fix it, or do some error recovery so that program execution can continue.

Guideline 1: Use exceptions to handle cases of abnormal function behavior.

Guideline 2: Use exceptions to handle cases of violated pre-conditions.

Guideline 3: Throw exceptions as soon as an error is known.

Guideline 4: Never assume the correctness of user input. Always check and possibly throw an exception if user input is wrong.

14.2 Assertions

Assertions are Boolean conditions inserted in code. An assertion states what you believe should be TRUE, according to program specifications, at the program point where you inserted an assertion. As such, assertions are a bridge between program code and requirements. If assertion evaluates to TRUE – the program works as expected; if it evaluates to FALSE – this means that an error occurred. Assertions play an important role in documenting and testing programs.

Usually, program requirements are defined externally to a program. The drawback is that it becomes so easy to change the program without updating the requirements document and vice versa. The power of assertions is that they coexist with the program code, making it easier to keep specifications up to date with evolving code and vice versa.

Assertions are conditions that express your intention as to what a correct program behavior should be. Macro assert is defined in <cassert.h>. Whenever assertion condition evaluates to 'false' – which means a runtime error – an error message is printed and program aborts. Therefore, assertions cannot be used to recover a program from the error situation.

Assertions are useful for debugging and also as program documentation. Before exceptions were introduced into programming languages, assertions often played the role of exceptions. At which program points should you insert assertions? Any program point that marks a meaningful stage in computation could be annotated with a suitable assertion. Any part of a code (e.g., class method) whose correct execution depends on the program state that can be formulated as a condition could be annotated with an assertion. Insert assertions into your program to test values of critical variables or function parameters to see if they have correct values during execution. In case they do not – print a meaningful error message that will help you localize an error during debugging. Do not remove these error tests from code even if your code works fine, as they will be useful in the future iterations when your program changes.

Checking assertions may affect performance, but assertions are only checked in a debugging mode. You can exclude assertions from a production version by including the line:

```
#define NDEBUG
```

Guideline 5: Do not write assertions for cases already catered for in exceptions.

Guideline 6: Write assertions to check any conditions that characterize a correct program behavior. Usually, such conditions are implied by the semantics of your program. In particular, post-conditions for functions can be checked by assertions.

For example:

```
// Return parent of input node
TNode* AST::getParent( const TNode* q )
{
// Pre-condition: Make sure input node is not NULL
        assert( q != NULL );

        TNode* p = q->parent;

// Post-condition: Parent can only be NULL, while or if
        assert( ( p == NULL ) || ( p->type == WHILE ) || ( p->type == IF ) );
        return p; }
```

Guideline 7: Write assertions whenever you feel they are useful for program documentation.

References:

Here are pointers where you can find exception basics as well as more advanced discussions of using exceptions and assertion in error handling:

C++ Tutorial http://cplus.about.com/library/weekly/aa122202a.htm

C++ Exceptions http://cis.stvincent.edu/carlsond/swdesign/except/except.html

15 Testing your Programs

Beyond certain code size and complexity, error-free programs do not exist. You test programs to uncover errors (or to assess the level of program reliability). Reliability requirements for industrial software are higher than for most of the course assignments in university courses. Students often say “we are done!” at first successful compilation and execution of a program for some simple input data. In industry, this is only the beginning and there is a long way before a program is considered finished. That is why in industrial development, testing takes up to 40% of the total project effort.

In this project, we expect program reliability close to industry standards. Allocate enough time for test planning and testing. Make unit testing an integral part of development. Use assertions to document your program and facilitate testing. Allocate time for integration testing and system testing whenever you have wrapped up a piece of work, at least at the end of each iteration. It is a common practice in successful companies to do integration and system testing on daily basis!

15.1 You Should Do the Following Types of Testing:

1. Unit testing (white-box testing): Testing of individual modules (functions, methods, classes). Try to spot as many errors as possible at the unit level – it will be more time consuming to do this during integration.
    a) It is responsibility of the developer to do unit testing. Make a habit to incorporate unit testing into your everyday programming.
    b) You may need to develop test drivers to simulate the environment of the unit under testing.
    c) Save test cases so that you can reuse them later when your program changes (regression testing).
    d) Use assertions to insert runtime checks into code. In case of error, print meaningful error message and terminate the program.
    e) Document the scope of unit testing that you have done.
    f) Make it possible to re-run unit testing after changes.
2. Integration testing: testing of a group of components that collectively achieve some higher-level function.
    a) This is primarily validation of component interfaces – does service provider and its clients have the same understanding of the interface operations? You have already tested separately Query Preprocessor and Query Evaluator – but do they work fine when you put them together?
    b) Send random samples of messages to components and validate messages received.
    c) Perform integration testing a couple of times in each iteration. Integration testing will show if you got the major decisions right. It may indicate possible miscommunication among team members – the earlier the better.
3. Validation testing: Checking if the system meets requirements
    a) Do validation testing at least once at the end of each iteration.

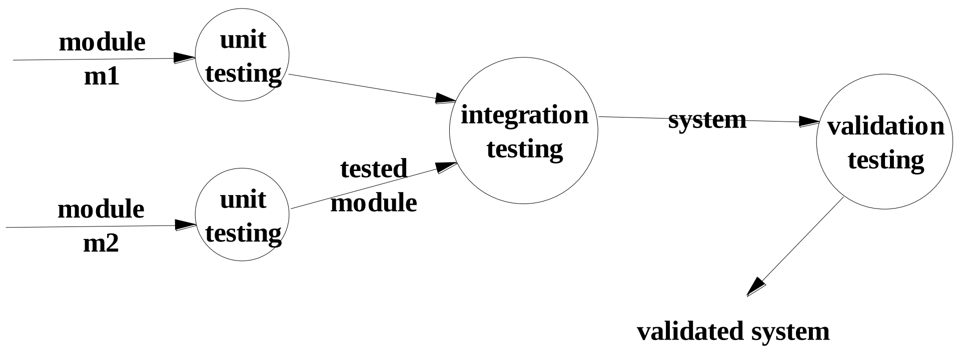


Figure 22. Testing activities

15.2 Test Plans and Test Case Documentation

An effective test case is one that breaks the program – testing is a destructive activity that should be approached with different mindset than development. Testing should be planned rather than done in ad hoc way. Validation and system testing can be planned as soon as requirement specifications have been completed. If your system passes validation and system testing – your know that you are done.

Test cases must be documented in a standard way and you should be able to repeat tests after changes (i.e., many times during system development and also during future maintenance). Test cases should be stored in a library for future use. Test library must be kept up to date with the evolving program.

Testing is often done by independent groups. So it is essential to document test cases in such a way that others can run them and check the results.

You will be required to produce test plans only for integration and validation testing, documenting each test case in standard was as follows:

Test Purpose: explain what you intend to test in this test case

Required Test Inputs: explain what program module (or the whole system) you test and what input must be fed to this test case

Expected Test Results: specify the results to be produced when you run this test case

Any Other Requirements: describe any other requirements for running this test case; for example, to run a test case for a program module in isolation from the rest of the system, you may need to implement a simulated environment to call the test case.

For unit testing, informally describe the scope of testing and provide samples.

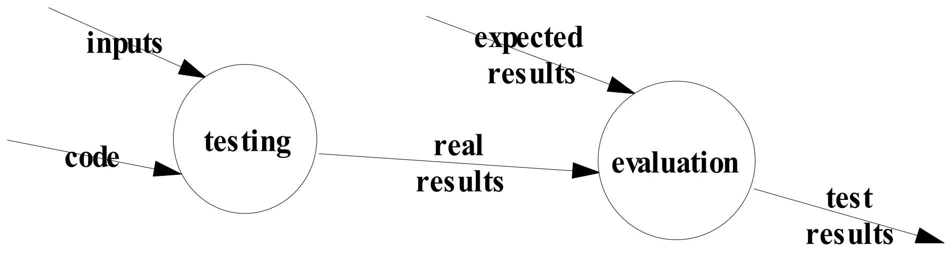


Figure 23. Running a test case and evaluating the results

16 Technical Tips

16.1 User Interface to SPA

User interface consists of three parts, namely source program entry, query entry and query result display. User interface will be very simple.

16.2 Parsing *SIMPLE*

You can use the technique described here to parse *SIMPLE* programs and *PQL* queries. For *PQL* queries, refer to section "Entering program queries (Group-*PQL*)" first.

Start with the following subset of *SIMPLE:*
program : procedure
procedure : 'procedure' proc_name '{' stmtLst '}'
stmtLst : stmt ';' (stmt ';')*
stmt : assign
assign : var_name '=' var_name | const_value
constant : INTEGER

You are free to choose parsing technique (for example, you may choose to implement parser in Perl). Here, we describe an easy way to write a predictive recursive descent parser. For each grammar symbol (such as program, procedure, etc.) have a procedure that attempts to recognize a corresponding program structure. Procedure GetToken () reads and returns a token. As the language rules are simple, we can determine what to do (i.e., which procedure to call) based on the next token (stored in variable next_token). Assume that source programs in *SIMPLE* are syntactically correct – **Group-PKB does not have to deal with error handling when parsing source programs**. In case of error – just stop parsing. Here is a pseudocode:

```
Match (token) {
        if (next_token = token)
                next_token = GetToken ()
        else
                Error ()
}

Program () {
        next_token = GetToken();
        Procedure () }

Procedure () {
        Match ('procedure');
        Match (proc_name);
        Match ('{');
        StmtLst ();
Match ('}');
}

StmtLst () {
        Stmt ();
        Match (';');
        if ( next_token = '}' )
                stop;
        else
                StmtLst ();
}
```

```
Stmt () {
        Match (var_name);
        Match (‘=’);
        Match (var_name or const_value);
}
```

Once you have implemented and tested a parser for the above subset of SIMPLE, you will need to extend it to parse expressions. You will find useful hints in Parsing Expressions by Recursive Descent by Theodore Norvell at http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm.

In case of error, your parser should print an error message and stop parsing.

Your parser should allow one to freely use spaces as separators in a program. For example, the parser should accept any of the following statements:

```
x=a+b;
x = a + b ;
while i{
        x=0;}
while i {
        x=0 ; }
```

16.3 Notes on AST

You will need to interlink AST nodes for ease of traversing an AST up and down. Each AST node should include at least the following three links:

firstChild link	rightSibling link	up link

You may find some extra links useful. Think about suitable representation of relationships ‘Parent’ and ‘Follows’. Should you have links for them in the AST or would you rather compute them during query evaluation based on the above three links? What are the trade-offs? Definitely the transitive closures ‘Follows*’ and ‘Parent*’ should be computed on demand during query evaluation.

16.4 Generating an AST during Parsing

Before you make your parser generate an AST for the source program, extend parser’s pseudocode with AST generation actions. Not only will this enhance your understanding of the AST generation logic before the actual implementation, but it will also help you convey to the PKB designers your needs with respect to parser’s interface to the AST. Here is a sample:

```
TNode* StmtLst () {
      TNode *curNode, *nextNode;
      curNode = Stmt ();
      Match (‘;’);
      if ( nextToken = ‘}’ )  return curNode;
      else {
        nextNode = StmtLst ();
        curNode -> setRightSib (nextNode);
        return curNode; }
}
TNode* Stmt () {
      Tnode *assign, *leftVar, *expr;
      assign = new (‘assign’);
      Match (var_name);
      leftVar = new (var_name);
```

```
        assign -> setFirstChild (leftVar);
        Match (‘=’);
        Match (var_name or const_value);
        expr = new (var_name or const_value);
        leftVar -> setRightSib (expr);
        return assign;
}
```

16.5 Creating Symbol Tables during Parsing

Your parser should store procedure and variable names in the symbol tables. All the references to procedures and variables in the AST, CFG and in other program design abstractions should be replaced by indices to the symbol tables. A procedure table (ProcTable) should contain procedure name, reference to the root of AST, reference to the CFG entry node, reference to the vector of modified variables, etc.

Index	procedure name	AST root	CFG entry	Modified	Used
1	First				

Table 3. ProcTable

Symbol tables will nicely integrate program design abstractions in the PKB. By consulting the ProcTable, you will be able to tell which procedures a program consist of, where are their respective ASTs and CFGs, etc.

Store information about which procedures call which other procedures (i.e., Calls relationship) in the CallsTable. You will not store relationship Calls*, rather you will compute Calls* information as needed, based on CallsTable.

16.6 What Information Should Be Stored in the PKB?

It is not practical to store all the program design abstractions specified above in the PKB. Some of the information will be computed “as needed” from the PKB, when you evaluate queries. Your PKB should contain: an AST, Calls, Modifies, Uses and CFG (Next relationship). Transitive closures of relationships will be computed “as needed” from PKB during query evaluation (the PKB API should provide suitable interface operations to do so). You will have to decide whether relationship Affects() should be pre-computed and stored in the PKB or rather also computed “as needed”.

What are the factors to consider and trade-offs in deciding if some information should be stored in PKB or computed “as needed”?

16.7 Relationships Modifies and Uses

To each procedure, statement and statement block (in while loop body, if-then and if-else), you will need to attach a corresponding set of variables modified and used.

Consider bit vector as a compact and easy to manipulate implementation of the relationships Modifies and Uses. As you will store program variables in the symbol table, each variable may be referred to by a unique number, an index in the symbol table. Represent relationship Modifies (s, v) as a bit vector in which i^{th} bit is 1 if variable with index i is modified in statement s; otherwise, the i^{th} bit is 0. Similarly, you will have a bit vector for relationship Uses (s, v).

16.8 Processing *PQL* Queries

Query processing subsystem consists of a Query Preprocessor, Query Evaluator and query result projector. Query Preprocessor validates a program query written in *PQL* and builds an internal data structure, so-called query tree. Query validation includes checking if references to design entities, attributes and relationships are used according with their definition in the program design model. The

Query Preprocessor consults the tables describing a program design model to validate queries. Refer to Section 13 "Table-driven Technique – Query Validation Example" for further details.

Query Evaluator traverses the query tree and consults the PKB in order to evaluate a query. Designing a general query evaluation mechanism that can answer any query described in the last section is a complicated task. The task becomes even more complex if you want to optimize query evaluation with respect to time and/or memory usage. In this project, you will design an evaluator only for certain types of program queries. If you try to implement some optimizations, your project will qualify for a higher grade.

Query result projector formats the query results in the format suitable for displaying.

16.8.1 Entering program queries (Group-PQL)

You can apply the technique described in section Parsing *SIMPLE* to parse *PQL* queries.

16.8.2 Query Preprocessor

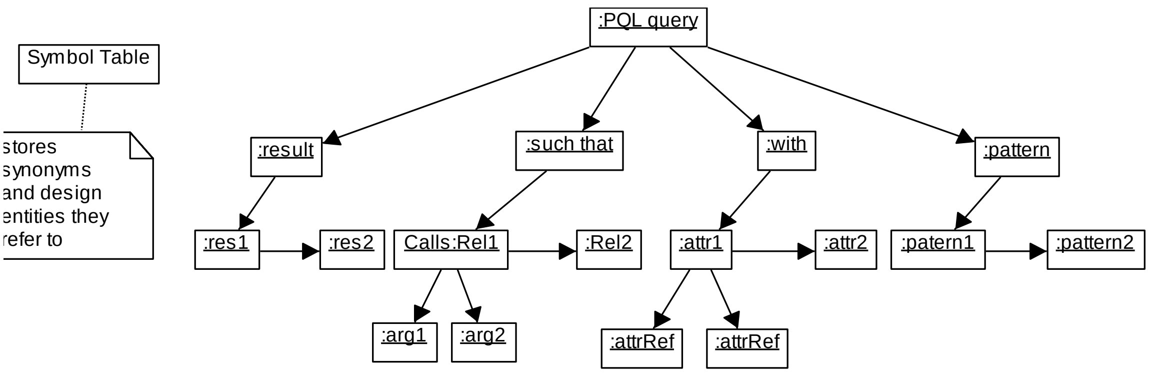


Figure 24. A query tree

The role of the Query Preprocessor is to validate a query and build query tree. Query validation means checking if all the references to program design entities, their attributes and relationships in a program query agree with the program design model definition. Refer to Section 13 "Table-driven Technique – Query Validation Example" for further details and design guidelines.

You need represent program queries in the form that is convenient for query evaluation (and eventually optimization). Textual representation is not suitable for that purpose (why?). Represent a query as a query tree. For example:

The overall structure of a query is represented in a tree form. Logic expressions in a query refer to design entities, their attributes and entity relationships of conceptual models of program design. The query tree can be traversed to evaluate a query.

Make your Query Preprocessor build the query tree. References to design entities, relationships and attributes in the query tree should be represented by references to entries in the program design model tables used by Query Preprocessor to validate queries.

16.8.3 Query evaluation and optimization

Query Evaluator may evaluate queries incrementally as follows. Having built a query tree, interpret query fragments contained in the **such that**, **with** and **pattern** clauses one by one. In that way, you divide the query evaluation process into a sequence of simpler evaluation steps. Evaluation of each query fragment involves calls to one or more interface operations to access information from the PKB. Evaluation of each query fragment produces an intermediate result that approximates the query result.

Here are some basic hints for implementing Query Evaluator and optimization:

First design the best possible Basic Query Evaluator (BQE) that can correctly evaluate any query without optimizations. Query involving many relationships must be evaluated incrementally, in steps. In the design of BQE pay attention to computing and representing intermediate query results. BQE should be prepared to employ various optimization strategies such as rearranging the order in which to evaluate relationships in a query and many others. You will have to experiment with various optimization strategies; that is why it is important that different optimization strategies can be plugged-into your BQE. BQE should not depend on any specific optimization strategy, but easily work with any optimization strategy you can think of.

The incremental strategy is relatively straightforward to implement but, without optimizations, query evaluation will be slow and will take much memory.

Of course, the size of the source program affects the efficiency of query evaluation. However, design decisions considering the PKB and Query Evaluator also have much to do with it.

The choice of data representation in the PKB affects performance of query evaluation. If you design your data structures (for AST, CFG, etc.) taking into account the ways you will need to compute information during query evaluation, your Query Evaluator will run faster. Let us say you have stored only forward links among your CFG nodes. Evaluation of the query that requires traversing the CFG backwards will be very time consuming. The way you store information about variables modified/used will have much impact on the evaluation time of queries that involve relationship Affects().

The main factors that have to do with effective query evaluation is the size of the intermediate result and the evaluation time. Optimizations should reduce the size of the intermediate result and/or the evaluation time. Changing the order in which you evaluate relationships in a query, as well as extra information about the source program may help you evaluate queries in more effective way. Assess various optimizations analytically before you implement them.

For example, it will take much longer to evaluate query:

assign a1, a2; while w;

Select a1 **such that** Follows (a1, a2) **and** Follows (a2, w)

than an equivalent query:

Select a1 **such that** Follows (a2, w) **and** Follows (a1, a2)

Generally, we should try to rearrange conditions so that we compute the most restrictive conditions first. By following this rule, intermediate sets of tuples are smaller and the query evaluation time is minimized.

Analysis of arguments of relationships can be also helpful. Consider the following query:

assign a; while w;

Select a **such that** Follow (a, w)

To evaluate this query, we could start by finding either assignments (‘assign’) or while loops (‘while’). If a program contains more assignments than while loops (most programs do), the query will evaluate faster if we find while loops first. Therefore, an optimized Query Evaluator could attempt to find while loops first and then compute assignments such that Follow (assign, while).

The above are examples of issues that you can take into account when optimizing queries.

Appendix A. Summary of *PQL* Grammar Rules

Lexical tokens are written in capital letters (e.g., LETTER, INTEGER, STRING). Keywords are between apostrophes (e.g., ‘procedure’). Non-terminals are in small letters.

<u>Meta symbols</u>:
a* -- repetition of a 0 or more times
a+ -- repetition of a 1 or more times
a | b -- a or b
brackets (and) are used for grouping

<u>Lexical rules</u>*:*
LETTER : A-Z | a-z -- capital or small letter
DIGIT : 0-9
IDENT : LETTER (LETTER | DIGIT | ‘#’)*
INTEGER : DIGIT+

<u>Auxiliary grammar rules:</u>
tuple : elem | ‘<’ elem (‘,’ elem)* ‘>’
elem : synonym | attrRef
synonym : IDENT
attrName : ‘procName’ | ‘varName’ | ‘value’ | ‘stmt#’
entRef : synonym | ‘_’ | ‘“’ IDENT ‘“’ | INTEGER
stmtRef : synonym | ‘_’ | INTEGER
lineRef : synonym | ‘_’ | INTEGER
design-entity : ‘procedure’ | ‘stmtLst’ | ‘stmt’ | ‘assign’ | ‘call’ | ‘while’ | ‘if’ | ‘variable’ |
‘constant’ | ‘prog_line’ | plus | minus | times

<u>Grammar rules for select clause:</u>
select-cl : declaration* ‘**Select**’ result-cl (suchthat-cl | with-cl | pattern-cl)*
declaration : design-entity synonym (‘,’ synonym)* ‘;’
result-cl : tuple | ‘BOOLEAN’

with-cl : ‘**with**’ attrCond
suchthat-cl : ‘**such that**’ relCond
pattern-cl : ‘**pattern**’ patternCond

attrCond : attrCompare (‘**and**’ attrCompare)*
attrCompare : ref ‘=’ ref
// left-hand-side and right-hand-side 'ref' must be of the same type (INTRGER or character string)
ref : attrRef | synonym | ‘”’ IDENT ‘”’ | INTEGER
// in the above, 'synonym' must be a synonym of prog_line
attrRef : synonym ‘.’ attrName

relCond : relRef (‘**and**’ relRef)*
relRef : ModifiesP | ModifiesS | UsesP | UsesS | Calls | CallsT |
Parent | ParentT | Follows | FollowsT | Next | NextT | Affects | AffectsT
ModifiesP : ‘Modifies’ ‘(’ entRef ‘,’ varRef ‘)’
ModifiesS : ‘Modifies’ ‘(’ stmtRef ‘,’ varRef ‘)’
UsesP : ‘Uses’ ‘(’ entRef ‘,’ varRef ‘)’
UsesS : ‘Uses’ ‘(’ stmtRef ‘,’ varRef ‘)’
Calls : ‘Calls’ ‘(’ entRef ‘,’ entRef ‘)’
CallsT : ‘Calls*’ ‘(’ entRef ‘,’ entRef ‘)’
Parent : ‘Parent’ ‘(’ stmtRef ‘,’ stmtRef ‘)’
ParentT : ‘Parent*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

```
Follows : ‘Follows’ ‘(’ stmtRef ‘,’ stmtRef ‘)’
FollowsT : ‘Follows*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’
Next : ‘Next’ ‘(’ lineRef ‘,’ lineRef ‘)’
NextT : ‘Next*’ ‘(’ lineRef ‘,’ lineRef ‘)’
Affects : ‘Affects’ ‘(’ stmtRef ‘,’ stmtRef ‘)’
AffectsT : ‘Affects*’ ‘(’ stmtRef ‘,’ stmtRef ‘)’

patternCond : pattern ( ‘and’ pattern )*
pattern : assign | while | if
varRef : synonym | ‘_’ | ‘“’ IDENT ‘“’
assign : synonym ‘(’ varRef ‘,’ expression-spec | ‘_’ ‘)’
expression-spec :            ‘“’ expr ‘“’| ‘_’ ‘“’ expr ‘“’ ‘_’
/* ‘synonym’ above must be of type ‘assign’
   expr must be a well-formed expression in SIMPLE
   refer to query examples in section 7.5 in the Project Handbook */

if : synonym ‘(’ varRef ‘,’ ‘_’ ‘,’ ‘_’ ‘)’
   // ‘synonym’ above must be of type ‘if’
while : synonym ‘(’ varRef ‘,’ ‘_’ ‘)’
   // ‘synonym’ above must be of type ‘while’
```

Summary of Type Checking Rules (So-called Static Semantics)

Any valid query must conform to additional rules, some of which cannot (or are difficult) to represent in BNF grammar:

1) All the synonyms used in a query must be declared.
2) A synonym listed in query results must be a synonym of a design entity that can be identified by name, value or statement/program line number. Otherwise, we cannot display a query result. Valid synonyms are ‘procedure’ | ‘stmtLst’ | ‘stmt’ | ‘assign’ | ‘call’ | ‘while’ | ‘if’ | ‘variable’ | ‘constant’| ‘prog_line’
   The query result is to be shown as follows:
- in case of procedures: a procName,
- in case of statements: an integer stmt#,
- in case of stmtLst: a stmt# of the first statement in the list,
- in case of variable: a varName,
- in case of constant: a value (integer), and
- in case of prog_line: a program line number (integer).
3) Arguments in relationships should be synonyms, ‘_’ and, depending on the relationship, integer (statement line numbers or program line numbers) or string (variable or procedure names). Relationship arguments should conform to program design abstraction models for SIMPLE, as defined in Section 6.
4) Please check conventions described in Section 7.2 (and summarized below), as they apply in addition to the above *PQL* core rules.
5) Underscore ‘_’ is a placeholder for an unconstrained synonym. Symbol ‘_’ can be only used when the context uniquely implies the type of the design entity denoted by ‘_’.
6) Under **with**-clause we can compare an attribute value and constant (integer or string, depending on the type of attribute) or two attribute values (provided they are of the same type).

Summary of Other *PQL* Rules:

1) A *PQL* query can contain any number of **such that, with** and **pattern** clauses. All the clauses may appear more than one time in a query, in any order, e.g.:

Select … **with** … **such that** … **with** … **with** … **pattern** … **such that** …

2) There is a default **and** between any two consecutive clauses. Therefore, we can swap or merge clauses without changing the meaning of the query.
3) A query result must satisfy all the query conditions in all the clauses. The existential quantifier is always implicit in *PQL* queries. That means, a query returns any result for which THERE EXISTS a combination of synonym instances satisfying the conditions specified in all the query conditions.
4) If the query result clause is a tuple, then tuple elements that satisfy all the query conditions at the same time are reported as query result.
5) Query with result BOOLEAN returns true if there is at least one combination of synonyms satisfying all the query conditions.
6) Spaces can be freely used and are meaningless (also multiple spaces).
7) You can make your own assumptions about any other details that have not been specified in the Handbook – such as case-sensitivity of identifiers/keywords.

Allowable Arguments of Relationships in Program Queries

In PQL queries, an argument in a relationship can be as follows:

1) A synonym of a valid design entity (valid according to a model of a given relationship, as defined in Handbook, Section 6).
2) A placeholder '_' (that is an unconstraint argument) provided it does not lead to ambiguity.
3) A character string in quotes, e.g., "xyz" can be used as an argument whenever an instance of the design entity can be identified by that string. When a relationship argument is a program, procedure, or variable then the string is interpreted as its name (should be NAME in quotes). When relationship argument is constant then string is interpreted as its value (the string should be INTEGER in quotes, e.g., "2").
4) An integer can be used as an argument whenever an instance of the design entity can be identified by integer. In the relationships Next and Next*, integer arguments mean program line numbers, and in Modifies, Uses, Parent, Parent*, Follows, Follows*, Affects, and Affects* an integer argument means statement number.
5) A synonym of prog_line can be used for arguments of type stmt. Then, the prog_line is interpreted as a statement number.
6) Synonyms of stmt, assign, call, while and if can appear in place of program lines in Next and Next* relationship. Statement numbers are then interpreted as program lines.
7) Synonyms of assign, call, while and if can be used as arguments of relationships in place of design entity stmt. E.g., Follows (a, w), where a is a synonym of assign and w is a synonym of while.

Appendix B: Summary of Design Abstractions

Relationship **Calls** (p, q) holds if procedure p directly calls q.

Calls* (p,q) holds if procedure p calls directly or indirectly q, that is:

Calls* (p, q) if:

Calls (p, q) or

Calls (p, p1) and Calls* (p1, q) for some procedure p1.

Relationship **Modifies** is defined as follows:

5. For an assignment a, Modifies (a, v) holds if variable v appears on the left hand side of a.
6. For a container statement s (i.e., ‘if’ or ‘while’), Modifies (s, v) holds if there is a statement s1 in the container such that Modifies (s1, v) holds.
7. For a procedure p, Modifies (p, v) holds if there is a statement s in p or in a procedure called (directly or indirectly) from p such that Modifies (s, v) holds.
8. For a procedure call statement s ‘call p’ Modifies (s, v) is defined in the same way as Modifies (p, v).

Relationship **Uses** is defined as follows:

1. For an assignment a, Uses (a, v) holds if variable v appears on the right hand side of a.
2. For a container statement s (i.e., ‘if’ or ‘while’), Uses (s, v) holds if v is a control variable of s (such as ‘while v’ or ‘if v’), or there is a statement s1 in the container such that Uses (s1, v) holds.
3. For a procedure p, Uses (p, v) holds if there is statement s in p or in a procedure called (directly or indirectly) from p such that Uses (s, v) holds.
4. For a procedure call statement s ‘call p’ Uses (s, v) is defined in the same way as Uses (p, v).

Abstract Syntax Grammar for *SIMPLE* (for AST)

Meta symbols: a+ means a list of 1 or more a’s; ‘|’ means or

program : procedure+
procedure : stmtLst
stmtLst : stmt+
stmt : assign | call | while | if
assign : variable expr
expr : plus | minus | times | ref
plus : expr expr
minus : expr expr
times : expr expr
ref : variable | constant
while: variable stmtLst
if : variable stmtLst stmtLst

Attributes and attribute value types:
procedure.procName, call.procName, variable.varName : NAME
constant.value : INTEGER
stmt.stmt# : INTEGER (line number of a given statement, see Figure 2)

Concrete Syntax Grammar for *SIMPLE* (for parsing)

Meta symbols:

```
a*      -- repetition 0 or more times of a
a+      -- repetition 1 or more times of a
a | b   -- a or b
brackets ( and ) are used for grouping
```

Lexical tokens:

```
LETTER : A-Z | a-z -- capital or small letter
DIGIT : 0-9
NAME : LETTER (LETTER | DIGIT)*  -- procedure names and variables are strings of letters, and
digits, starting with a letter
INTEGER : DIGIT+  -- constants are sequences of digits
```

Grammar rules:

```
program : procedure+
procedure : ‘procedure’ proc_name ‘{‘ stmtLst ‘}’
stmtLst : stmt+
stmt : call  |  while | if | assign
call : ‘call’ proc_name ‘;’
while : ‘while’ var_name ‘{‘ stmtLst ‘}’
if : ‘if’ var_name ‘then’ ‘{‘ stmtLst ‘}’ ‘else’ ‘{‘ stmtLst ‘}’
assign : var_name ‘=’ expr ‘;’
expr : expr ‘+’ term | expr ‘-’ term | term
term : term ‘*’ factor | factor
factor : var_name | const_value | ‘(’ expr ‘)’
var_name : NAME
proc_name : NAME
const_value : INTEGER
```

For any two statements s1 and s2, the relationship **Parent** (s1, s2) holds if s2 is directly nested in s1.

Therefore, s1 must be a ‘container’ statement. In *SIMPLE* there are two containers, namely ‘while’ and ‘if’. In terms of AST, Parent (s1, s2) holds if s2 is a direct child of stmtLst who in turn is a direct child of the AST node labeled with the container name (‘while’ or ‘if’).

Relationship **Parent*** is the transitive closure of relationship ‘Parent’, i.e.,

Parent* (s1, s2) if:

  Parent (s1, s2) or

  Parent (s1, s) and Parent* (s, s2) for some statement s.

For any two statements s1 and s2 relationship **Follows** (s1, s2) holds if s2 appears in program text directly after s1 at the same nesting level, and s1 and s2 belong to the same statement list (stmtLst). In terms of AST, Follows (s1, s2) holds if s2 is the direct right sibling of s1.

Follows* is the transitive closure of ‘Follows’, i.e.,

Follows* (s1, s2) if:
  Follows (s1, s2) or
  Follows (s1, s) and Follows* (s, s2) for some statement s.

Let n_1 and n_2 be program lines.

Relationship **Next**(n_1, n_2), holds if n_1 and n_2 are in the same procedure, and n_2 can be executed immediately after n1 in some program execution sequence.

The relationship **Next*** is the transitive closure of relationship ‘Next’:

Next*(n_1, n_k) if
Next (n_1, n_k) or
Next(n_1, n) and Next*(n, n_k) for some program line n.

A **control flow path** is any sequence of program lines (n_1, …, n_k)
such that Next (n_i, n_{i+1}) for i = 1, …, k-1

Let a1 and a2 be two assignment statements such that a1 modifies value of variable v and a2 uses the value of variable v.

Affects (a1, a2) holds if a1 and a2 are in the same procedure, and there is a control flow path from a1 to a2 (i.e., Next* (a1, a2)) such that v is not modified (as defined by Modifies relationship) in any assignment or procedure call statement on that path (excluding a1 and a2).

Let a1 and a2 be two assignment statements.

Affects* (a1, a2) holds if a1 and a2 are in the same procedure, and a1 has either direct or indirect impact on a2, i.e.:

Affects* (a1, a2) if

Affects (a1, a2) or

Affects (a1, a) and Affects* (a, a2) for some assignment statement a.

--- The End ---