PQL: A language for specifying abstract program views

Stan Jarzabek
Department of Information Systems and Computer Science
National University of Singapore

Abstract

A program query language, *PQL* for short, described in this paper is a source language-independent notation to specify program queries and program views. We use *PQL* as an interface to Static Program Analyzers (SPA), interactive tools that enhance program understanding by answering queries about programs. In *PQL*, we can query on global program design as well as search for detail code patterns. Program queries and patterns supported by other notations described in literature and those supported by commercial tools known to the author, can be written simply and naturally in *PQL*. Being driven by a conceptual program model and based on SQL and attribute abstract syntax grammar concepts, *PQL* is a high-level and intuitive notation for querying programs. Program modeling and *PQL* notations described in the paper form a basis for an SPA generation system.

I. Introduction

With maintenance consuming increasing share of computing costs, attention is drawn to methods and tools for program understanding. During program maintenance, programmers often extract program views that are relevant to a maintenance task in hand. However, in a huge, undocumented program it may not be easy to find program views in question. Extracting relevant program views manually is time consuming and may lead to errors. A Static Program Analyzer (SPA) extracts program views automatically (Fig. 1).

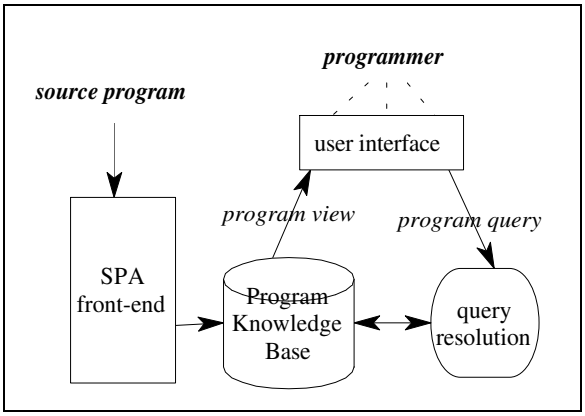

Fig. 1 Architecture of a Static Program Analyzer

An SPA front-end parses an input program into an internal representation and stores it in a Program Knowledge Base (PKB). A user interface component allows a programmer to input program queries and view query results. Usually, we design an SPA in the following steps [10]:

1. identify a class of program queries to be answered,
2. model program information that is needed to answer queries,
3. define physical representation for programs in the PKB; common representations include a relational database [4,14], object-oriented database [1,12], attributes syntax tree [7], dependency graph [12,15] and a hybrid PKB [9,10,11],
4. design an SPA front-end to generate the PKB from source programs,
5. decide how the programmer will enter queries,
6. design the query resolution mechanism,
7. design program view projector to display query results.

A simple solution to step 5 is to implement a fixed set of program queries for a programmer to use. A better solution is to provide means for a programmer to specify queries. We designed a general-purpose ***p***rogram ***q***uery ***l***anguage, *PQL* for short, to write program queries. *PQL* has the following unique features:

1. *High expressive power and simplicity.* In *PQL*, we can
    - query on global system-level design,
    - search for syntactic program patterns,
    - constrain syntactic patterns with semantic conditions,
    - combine global queries with code patterns using the same notational conventions.

    PQL queries are natural, as we write them in terms of an explicit conceptual program model.
2. *Hierarchical queries*. Complex program views can be expressed in terms of simpler ones. This feature makes it possible to formulate queries through levels of abstraction leading from code structures up to the level of program conceptualizations.
3. *Language-independence of PQL.* Same program modeling conventions are used independently of a source language. *PQL* queries are driven by the program model.
4. Conceptual program models and *PQL* described in this paper facilitate the systematic design scenario for Static Program Analyzers [10]. The objective of this scenario is to better link tool capabilities to the needs of tool users, and to design tools in a flexible and generic way.

We compared *PQL*'s expressive power with other notations described in literature [1,4,7,12,14,16] and with program views supported by commercial tools [19,21]. Comparison is favorable for *PQL.* All types of queries that we found in these sources can be expressed in *PQL.* In the remaining part of the paper, we describe related work and explain how we model programs. Then we describe the *PQL* and its implementations.

II. Related work

Examples of SPAs that store program information in a relational database include CIA [4] and OMEGA [14]. The advantage of using a relational database is that program information can be queried using SQL [5]. CIA stores only global descriptions of programs. Experiences with OMEGA show that if we store too much detail information about programs in a relational database, the performance of SPA may be poor. REFINE [1] stores programs as syntax trees in an object-oriented database. *PQL* described in this paper has similar expressive power to REFINE's query language. GENOA [7] uses abstract syntax trees and provides specification methods and algorithms that allow one to traverse syntax trees to derive any new type of information from them. Attribute evaluation mechanism has been extended from single to multiple, inter-related syntax trees [9]. Using these extensions, we can analyze and query families of programs [10,11]. A notation for C code pattern specification is described in [16].

III. Conceptual program models

In this section, we describe three components of a program model: a global program design model, program structure model and detail program design model. All three models use the same notational conventions of an extended OMT [18]. To facilitate explanations, we introduce a simple source language, L, with procedures, global and local variables. Procedures are grouped into modules. We use L to explain program modeling rules and queries in *PQL.*

A. Global program design model

Suppose we want to support program queries listed in box 1.

Q1. Which file contains module "Stack"?
Q2. Which procedures are declared in module "Stack"?
Q3. Which global variables have their values modified in procedure "Push"?
Q4. Which procedures are called from "Push"?
Q5. Find procedures that are called from more than 10 other procedures

Box 1. Global program queries.

The global design model (Fig. 2) shows *design entities* (in rectangular boxes), their attributes (above the entity box) and entity relationships. Dots stand for 'many' connectivity in a relationship link. The meaning of a relationship link is clarified by a role name attached to a link. Relationship 'Calls' describes direct procedure calls, while 'Calls*' is a transitive closure of Call: Calls* (p, q) **iff** Calls (p, p1) **and** Calls* (p1, q) for some p1.

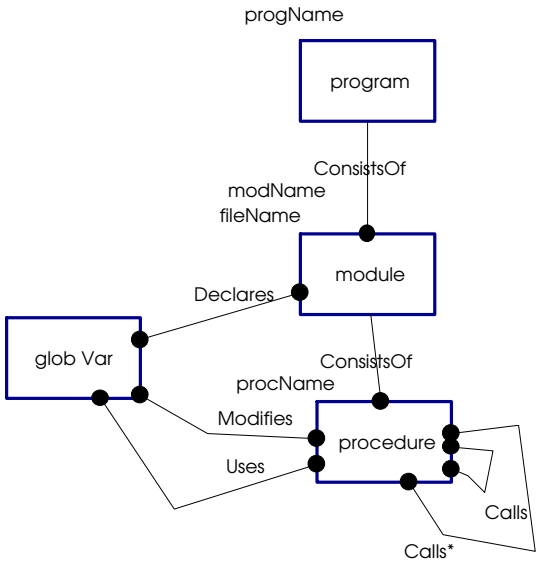

Fig. 2. A global program design model.

With the information model of Fig. 2, we can identify some other useful queries such as "Which procedures that call Push modify global variable X?".

B. Program structure model

Suppose now that, in addition to global queries, we wish to search for code patterns:

Q6. Find assignment statements where variable X appears on the LHS.
Q7. Find statements that contain sub-expression X*(Y+Z).
Q8. Find three while-do loops nested one in another.
Q9. Find all program statements that modify global variable X and use a global variable Y at the same time.

Box 2. Queries about program structure

To answer these and other queries about program structure, we need to access program information at the detail level. The required program information is best modeled by an abstract syntax grammar and then represented by syntax trees. A grammar in graphical form is given in Fig. 3a.

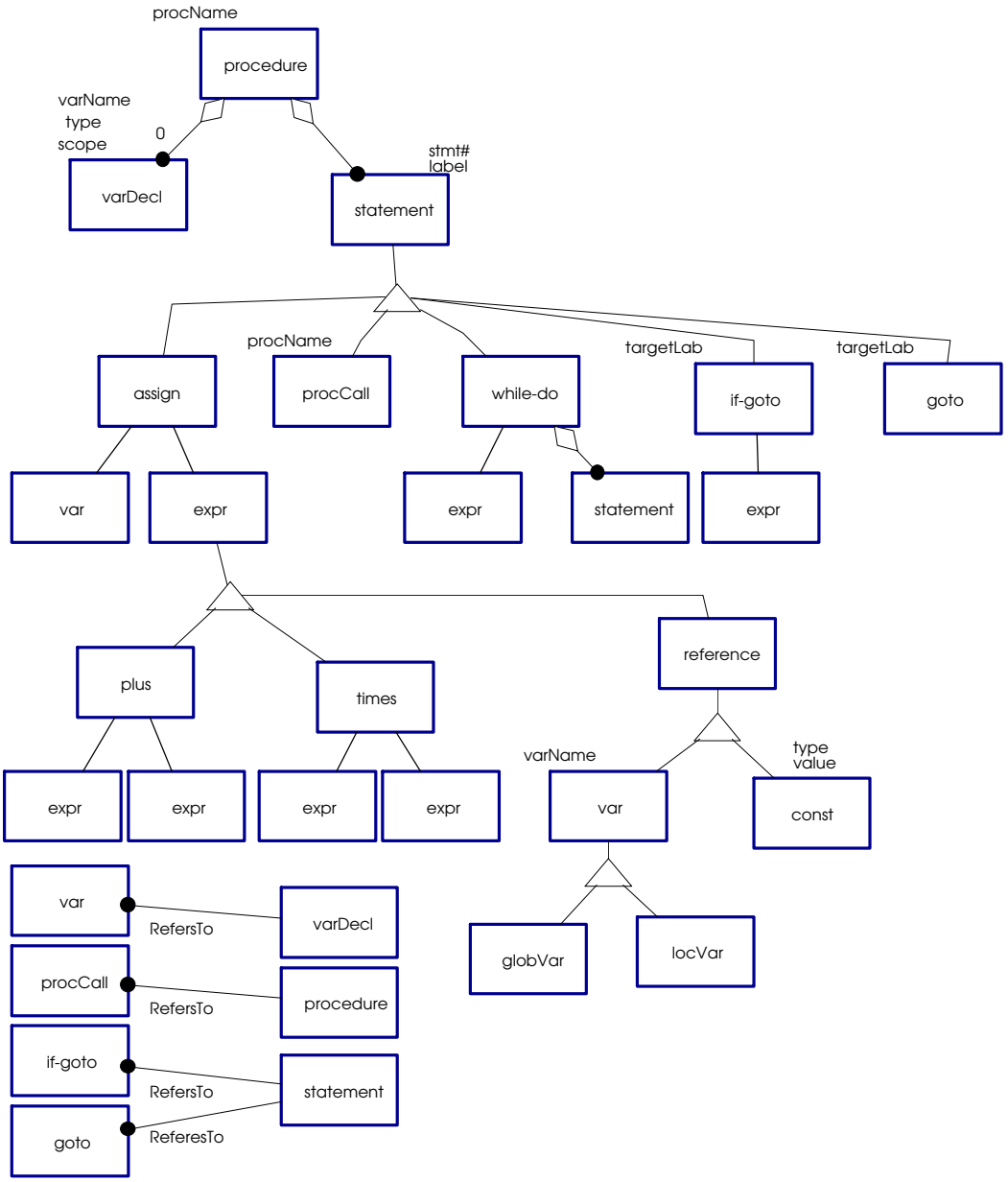

Fig. 3a. Program structure model.

We build a program structure model using similar conventions as those we used in the global design model. Grammar symbols (in boxes) represent *syntax entities*. Syntax entities are classified using IsA relationship (a triangle with general entity above and specialized entities below). Tuples of elements are identified by 1-to-many aggregation relationship link. There are two types of tuples, namely a tuple of locally declared variables (connectivity 0 above the 'var' box indicates that this tuple may be empty) and a tuple of statements (this tuple must contain at least one statement). A family of relationships 'RefersTo' complement the program structure definition. For example, relationship 'RefersTo' between 'var' and 'varDecl' indicates that each reference to a variable is liked to the corresponding declaration (e.g., an entry in the symbol table). Attributes (displayed above the syntax entity box) specify types of

information that must be available at syntax tree nodes. (At the modeling stage, we are not concerned with how this information is actually computed or stored.) Attributes assigned to a parent apply to all its children. Box 3 shows the grammar in predicate form. Code patterns in *PQL* queries are expressed using predicate grammar rules.

Abstract syntax grammar rules:
1. procedure (<varDecl*>, <statement+>)
2. assign , procCall , while-do, if-goto, goto IsA statement
3. assign (var, expr)
4. plus, times, reference IsA expr
5. plus, times (expr, expr)
6. var, const IsA reference
7. globVar, locVar IsA var
8. while-do (expr, <statement+>)
9. if-goto (expr)

Relationships:
RefersTo (var, varDecl) (n:1)
RefersTo (procCall, procedure) (n:1)
RefersTo (if-goto, statement) (n:1)
RefersTo (goto, statement) (n:1)

Attribute declarations:
procName : procedure, procCall
varName: varDecl, var
type, scope : varDecl
label, stmt# : statement
targetLab : if-goto, goto
type, value : const

Box 3. Abstract syntax grammar for L.

We extend the program structure model with additional classifications of syntax entities and relationships describing program structure. Later on we shall find them useful in formulating program queries.

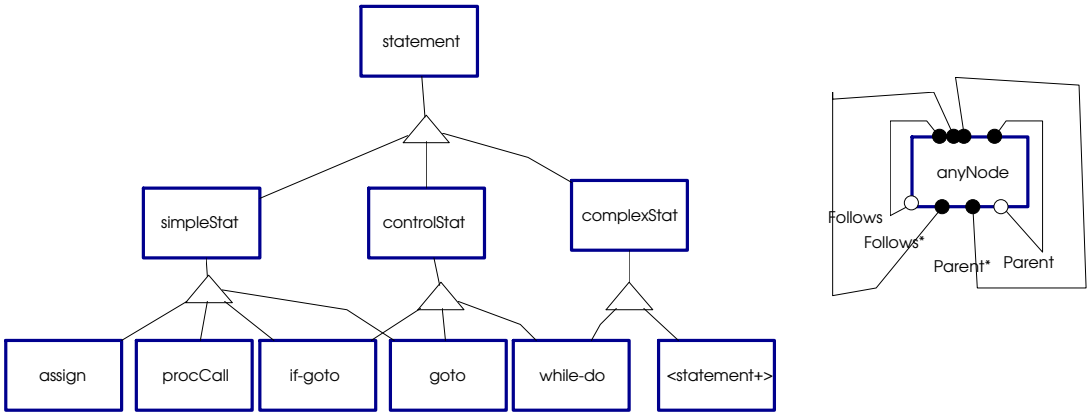

Fig. 3b. Extensions to the basic program structure model.

Entities may participate in more than one IsA classification. Entity 'anyNode' represents syntax tree nodes. For any two nodes n1 and n2 in a syntax tree, relationship Parent (n1, n2)

holds if n1 is the direct parent node of n2. Parent* is a transitive closure of relationship Parent, i.e., Parent* (n1, n2) if there is node x such that Parent (n1, x) and Parent* (x, n2). Similarly, relationship Follows (n1, n2) holds if n2 appears next to the right of n1 in a syntax tree. Follows* is a transitive closure of Follows, i.e., Follows* (n1, n2) if there is node x such that Follows (n1, x) and Follows* (x, n2).

C. Detail program design model

Suppose we want to answer the following questions:

Q10. Is there a control path from statement #20 to statement #620?
Q11. Which assignments that modify variable X affect value of X at statement #120?
Q12. Which program statements affect (directly or indirectly) value of X at statement #120?
Q13. If I change X:=5 to X:=10 in statement #20, which other program statements can be affected?
Q14. Find all assignments to variable X such that value of X is subsequently re-assigned a value recursively in an assignment statement that is nested inside two loops.
Q15. Find "dead code" in procedure "Parse".

Box 4. Detail program queries.

To address such queries, we extent our program model with detail program design information, as illustrated in Fig. 4.

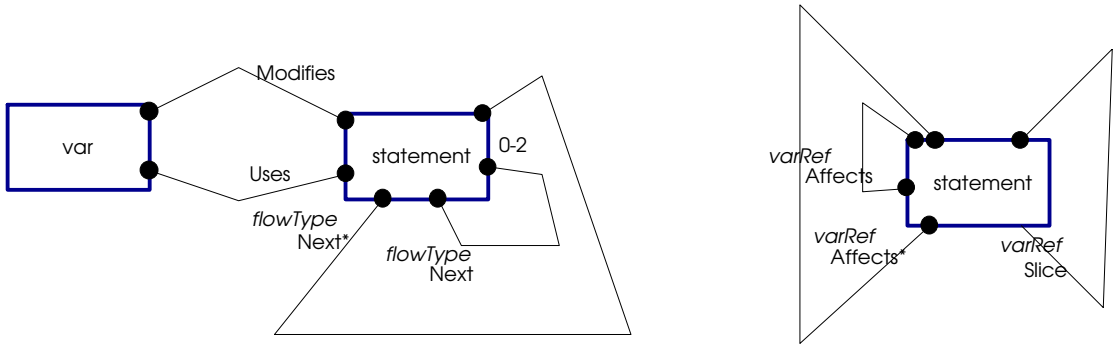

Fig. 4. Detail program design model.

Relationship Next (s1, s2, flowType) involving two program statements s1 and s2 holds if s2 can be executed immediately after s1 in some program execution. Relationship Next has an attribute *flowType* that we treat as the third relationship argument. This attribute describes the type of control flow:

sequential - if s2 appears after s1 and is executed upon normal completion of s1,

true - conditional control transfer that occurs when a condition in s1 is satisfied,

false - conditional control transfer that occurs when a condition in s1 is not satisfied,

loop-exit - loop exit

loop-iteration - starting a new loop iteration.

Some languages provide loop exits (such as break or continue in C) and possibility of entering a procedure body without actually calling the procedure ('flow-through' in COBOL). They all form different types of control flow that are recorded in attribute *flowType*.

Relationship Next* is a transitive closure of relationship Next, i.e., Next*(s1,s2,ft1) iff there is statement x such that Next(s1,x,ft2) and Next*(x,s2,ft3). Control flow types are irrelevant to Next*. Relationship Next* describes possible computational paths through procedure statements and complements relationship Calls in a global design model that describes possible sequences of procedure activation. For a given statement s, relationships Modifies and Uses define sets of variables modified and used in s, respectively. Attribute *varRef* is a reference to a variable (syntax entity 'var'). For given statements s1, s2 and variable reference 'v' relationship Affects(s1, s2, v) holds, if value of 'v' at s1 can be actually used when 'v' is referenced at s2 (i.e., there must be a computational path from s1 to s2 on which 'v' is not modified). Like in case of an attribute of relationship Next, we include attribute *varRef* of relationship Affects as the third argument of this relationship (which, in fact, is a ternary relationship). Relationship Affects*(s1, s2, v) identifies all the statements affected by value of 'v' at s1. A slice on variable 'v' at statement s1, Slice(s1, s2, v), contains all the statements s2 that contribute to value of 'v' at s1.

When building a program model, we identify types of program information that we shall need to answer program queries. We are not concerned with issues of how modeled information will be actually stored nor how it will be computed (of course, we should not model the information that is not computable). For example, some of the attributes may be persistent (i.e., pre-computed by the SPA front-end) and have their values stored directly at the syntax tree node or in a database, while others - computed on demand during query resolution. Similarly, relationships can be stored in the database, implemented as tree node attributes or computed on demand, depending on convenience and whether simplicity or efficiency of the solution is a factor. A reasonable solution, in case of the discussed program model, would be to make relationships Next, Modifies, Uses and Affects persistent and to compute Next*, Slice and Affects* on demand. These decisions belong to implementation domain and are of concern when we design a physical PKB and an SPA front-end, but not during conceptual modeling of program information.

D. Attribute domains

Attribute domains are simple or complex. Simple domains include strings (STR) and string subsets, integers (INT), reals (REAL) and booleans (BOOLEAN). String subsets are specified by regular expressions. An attribute domain may also be of type "reference to entity instance", e.g., attribute *varRef* of relationships Affect and Affects* is such a reference. Complex domains are defined by (abstract syntax) grammars [9]. Domains such as enumerations, tuples and record structures can be created in that way. All semantic domains are defined as abstract data types, hence they come together with operations for creating and manipulating their values.

The following are examples of attribute domains:

```
IDENT : [A-Za-z]*[0-9A-Za-z]  // a subset of STR defined by a regular expression in
LEX notation
int, real IsA TYPE  // this declaration defines enumeration domain
PROC : @procedure // a reference to instance of entity procedure
PROCS : <@procedure*> // a tuple of references to procedures
VAR : @var
LOCAL, global IsA SCOPE
LOCAL : @procedure
if-true-goto,  if-false,  goto,  loop-exit , loop-iteration IsA FLOW_TYPE
```

The following are attribute declarations for our program model:

```
IDENT progName : program
IDENT modName : module
STR fileName : module
IDENT procName : procedure, procCall
IDENT varName : var, varDecl
TYPE type : varDecl, const
SCOPE scope : varDecl
IDENT label : statement
INT stmt# : statement
IDENT targetLab : if-goto, goto
FLOW_TYPE flowType : Next (statement, statement), Next* (statement, statement)
VAR varRef : Affects (statement, statement), Affects* (statement, statement),
Slice (statement, statement)
```

Attributes *flowType* and *varRef* in last two declarations are attached to relationships.

E. The semantics of a program model

The meaning of relationships and attributes is not formally defined within the program model. Usually, the meaning of a program model is operationally defined by actions that derive information from source programs. Inferring the semantics of a program model from actions embedded in the SPA front-end code is not easy. It is, however, possible to specify the

meaning of program information in descriptive way and in terms of the conceptual program model, rather than in terms of physical PKB schema. This is done by separation of syntax tree construction from other program analysis actions. Program model derivation rules can then be described as attribute equations attached to abstract syntax grammar production rules. We refer the reader to other sources for details [8,9].

IV. Querying programs with *PQL*

A. Sets and tuples in *PQL*

Each program model entity represents a set of its instances. Relationship signature (e.g., Calls (procedure, procedure)) represents a subset of pairs of entity instances that are involved in a relationship. In *PQL*, tuples are typed and may be of fixed or unspecified length. Tuples may involve entity instances as well as attribute values. Here are examples:

<assign, while-do, procCall> - a set of triples of instances of entities assign, while-do and procCall, respectively,

<assign, procedure.procName> - a set of pairs fist of which is an instance of entity assign and second one is a procedure name,

<statement*> - a set of tuples of 0 or more statements,

<statement+> - a set of tuples of 1 or more statements,

<statement 3-7> - a set of tuples of 3 to 7statements.

A *PQL* query specifies elements to be retrieved. Elements may be entity instances, attribute values, tuples or a BOOLEAN value. Here are examples:

Select procedure

Select procedure.procName

Select <assign, procedure, globVar.varName>

Select BOOLEAN

In most situations, we wish to select a certain subset of elements rather the whole set. We constrain properties of the target subset by specifying conditions to be satisfied by its elements. Conditions are expressed in terms of:

a) entity instances, attribute values and constants,

b) participation of entities in relationships,

c) code patterns.

The following is a general format of a program query:

select-cl ::= [name-cl] declaration* **Select** result (with-cl | suchthat-cl)* [from-cl] [pattern-cl]*
cond-cl ::= (with-cl | suchthat-cl)*

Clauses in rectangular brackets are optional. Star '*' means repetition 0 or more times. Stepping from left, the *name*-cl gives a name to a retrieved program view. Declarations introduce synonyms for entities. Synonyms can be used in the remaining part of the query to mean a corresponding entity. The *result* specifies a program view to be produced. A program view is a subset of entity instances, a subset of tuples, a subset of entity attribute values or a BOOLEAN value. In the *with*-cl we constrain attribute values (e.g., procedure.procName="Parse"). The *suchthat*-cl, specifies conditions in terms of relationship participation. The *from*-cl restricts the scope of program to be searched for code patterns and *pattern*-cl describes code patterns to be searched for. Both *from*-cl and *pattern*-cl may include *cond*-cl, if necessary. We explain program queries by examples.

B. Querying global program design

We start by writing global program queries in *PQL* (see box 1, section III A).

Q1. Which file contains module "Stack"?

Select module.fileName **with** module.modName="Stack"

Explanations: keywords are in bold. Dot '.' notation means reference to the attribute value of an entity.

Q2. Which procedures are declared in module "Stack"?

Select procedure **such that** ConsistsOf (module, procedure) **with**
module.modName="Stack"

Q3. Which global variables have their values modified in procedure "Push"?

Select globVar **such that** Modifies (procedure, globVar) **with** procedure.procName=
"Push"

Q4. Which procedures are called from "Push"?

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

Explanation: variables p and q are synonyms of entity procedure.

Q5. Find procedures that are called from more than 10 procedures

procedure p, q
Select q **such that** Calls (p, q>10)

Explanation: constraint ">10" applies to the number of occurrences of a procedure as the second argument in relationship 'Calls'. First argument is unconstrained.

Q5a. Find procedures that are called from at least one other procedure.

procedure p, q

Select q **such that exists** [p **such that** Calls (p,q) **with** p≠q]

Explanation: notice use of existential quantifier. Inequality p≠q means that p and q must represent different instances of entity procedure.

Select q such that (not Calls (q,q) and Calls (p,q>0)) or (Calls (q,q) and Calls (p,q>1))

Explanation: the same query without existential quantifier.

Q5b. Find procedures that call all procedures

procedure p, q

Select p **such that forall** [q: Calls (p, q)]

or

Select p **such that** Calls (p=CARD(procedure), q)

Explanations: the same query without **forall** quantifier. Built-in function CARD returns the number of instances of an argument entity, in this case procedure.

Q5c. Find procedures that call only the procedure "Parse".

Select p **such that** Calls (p=1, q1) **and** Calls (p, q) **with** q.procName="Parse"

C. Specifying program patterns

We specify program patterns based on program structure model (Fig. 3a and 3b), using predicate notation (box 3). For example, pattern assign (var, expr) with unconstrained arguments is matched by all assignment statements. Notationally, syntax patterns are to relational conditions in global queries. Patterns are specified in **pattern** clause and can be further constrained by conditions listed in this clause.

Q6. Find assignment statements where variable X appears on the LHS.

Select assign **pattern** assign (var, _) **with** var.varName="X"

Explanation: underscore (_) stands for a free, unconstrained entity.

Q6a. Find assignment statements from procedure "Parse" where global variable X appears on the LHS.

procedure p

Select assign **from** p **with** p.procName="Parse"

pattern assign (globVar, _) **with** globVar.varName="X"

Explanation: here we use **from** clause to restrict the scope of searching for patterns.

Q7. Find statements that contain sub-expression X*(Y+Z).

Select assign **pattern** assign (_, expr) **such that** Parent* (expr, X*(Y+Z))

Explanation: infix expression X*(Y+Z) is used for convenience. It is translated into equivalent predicate form: times(v1,plus(v2, v3)) **with** v1.varName="X" **and** v2.varName="Y" **and** v3.varName="Z"

or without nested patterns:

assign (var, times) **and** times (v1, plus) **and** plus (v2, v3) **with** v1.varName="X" **and** v2.varName="Y" **and** v3.varName="Z"

Q8. Find three while-do loops nested one in another.

while-do s1, s2, s3

Select <s1, s2, s3> **such that** Parent* (s1, s2) **and** Parent* (s2, s3)

Q9. Find all program statements that modify global variable X and use a global variable Y at the same time.

globVar gX, gY; statement s; procedure p

Select s **from** p **such that** Modifies (p, gX) **and** Uses (p, gY)

with gX.varName="X" **and** gY.varName="Y"

such that Modifies (s, gX) **and** Uses (s, gY)

with gX.varName="X" **and** gY.varName="Y"

Explanation: this query retrieves all the statements (assignments, procedure calls and complex statements) that are known to modify global X and use global Y. (Notice that in case of aliases this query is stronger than just asking about assignments where global variable X appears on the LHS.) We use global model (Fig. 2) and **from**-clause to optimize query resolution: only procedures that are known to modify global X and use global Y are inspected in the search for patterns.

D. Querying detail program design

Q10. Is there a control path from statement #20 to statement #620 in procedure "Parse"?

statement s1, s2; procedure p

Select BOOLEAN **from** p **with** p.procName="Parse"

such that Next* (s1, s2) **with** s1.stmt#=20 **and** s2.stmt#=625

Explanation: the answer is determined based on the truth of **such that** condition.

Q11. Which assignments that modify variable X affect value of X at statement #120?

assign a; statement s

Select a **such that** Modifies (a, var) **and** Affects (a, s, var) **with** var.varName="X" **and** s.stmt#=120

Q12. Which program statements affect (directly or indirectly) value of X at statement #120?

statement s1, s2

Select s1 **such that** Slice (s1, s2, var) **with** s2.stmt#=120 **and** var.varName="X"

Q13. If I change X:=5 to X:=10 in statement #20, which other program statements can be affected?

assign a; statement s

Select s **such that** Affects* (a, s, var) **with** a.stmt#=20 **and** var.varName="X"

Q14. Find all assignments to variable X such that value of X is subsequently re-assigned a value recursively in an assignment statement that is nested inside two loops.

assign a1, a2; while-do w1, w2; expr e

Select a2 **pattern** a1 (var, expr) **and** a2 (var, e) **such that** Parent* (e, var)

with var.varName="X"

such that Affects (a1, a2, var) **and** Parent* (w2, a2) **and** Parent* (w1, w2)

Q15. Find "dead code" in procedure "Parse".

statement s1, s2; procedure p

Select s2 **from** p **with** p.procName="Parse"

such that not exists [s1 **such that** Entry (s1, p) **and** Next* (s1, s2)]

Explanation: Entry (statement, procedure) is a predicate that identifies the entry points to a procedure. Statements that are not reachable from the procedure entry point (in terms of the control flow) form dead code which is never executed.

E. Specifying reverse engineering transformations with hierarchical program views

In *PQL*, we can systematically progress from low level code views to higher levels of program understanding. High-level program views can be hierarchically built out of already constructed, simpler ones. Program views are named and typed (Table 1). A view can be included in the definition of any other view by name, as long there is no type conflict at the point of inclusion. Built-in operations on sets and tuples (Table 2) are helpful in constructing hierarchical program views.

PQL **query**	**type of a program view**
Select procedure	a set of procedures
procedure p1, p2 **Select** <p1, p2>	set of pairs of procedures
Select <assign, procCall, while-do>	triples of statements of specified types

Table 1. Types of program views.

Operations on sets and tuples	**the meaning**
int CARD (s)	returns the number of elements in s
bool IS-IN (e, s)	TRUE, if e is in s; otherwise FALSE
bool NOT-IN (e, s)	TRUE, if e is not in s; otherwise FALSE
bool IS-EMPTY (s)	TRUE, if s empty; otherwise FALSE
bool SUB-SET (s1, s2)	TRUE, if each element from s1 is in s2; otherwise FALSE
Other operations on tuples	
bool FOLLOWS (e1, e2, t)	TRUE, if e2 follows directly e1 in tuple t; otherwise FALSE
bool FOLLOWS* (e1, e2, t)	TRUE, if e2 follows indirectly e1 in tuple t; otherwise FALSE
bool SUB-SEQ (t1, t2)	TRUE, if t1 appears in t2; otherwise, FALSE
FIRST (e, t)	TRUE, if e is the first element of tuple t; otherwise FALSE
LAST (e, t)	TRUE, if e is the last element of tuple t; otherwise FALSE

Table 2. Operations on sets and tuples.

Here is an example of a hierarchical program view definition:

view use-X
Select procedure **such that** Uses (procedure, globVar) **with** globVar.varName="X"

view mod-X
Select procedure **such that** Modifies (procedure, globVar) **with** globVar.varName="X"

view use-mod-X
Select procedure **such that** IS-IN (procedure, use-X) **and** IS-IN (procedure, mod-X)

view ref-X
Select procedure **such that** IS-IN (procedure, use-X) **or** IS-IN (procedure, mod-X)

Hierarchical views are particularly useful in defining reverse engineering transformations. In case of big software systems, reverse engineering must be selective, otherwise recovered designs are too complex to be useful. In *PQL*, we can specify filters to be applied to recovered information to obtain more focused program views. Furthermore, with hierarchical views, filters can be applied in steps and each recovered view - inspected by a programmer. We find programmer's involvement invaluable in directing the reverse engineering process. Reverse engineering of views such as structure charts, structure chart slices and procedure interfaces can be described in *PQL*. We start by describing a structure chart for the whole system:

view structure-chart

procedure p1, p2

Select <p1, p2> **such that** Calls (p1, p2)

In case of a big system, this view may consist of thousands of interrelated procedures. We might want to compute a slice of the structure chart showing, for example, only those procedures that refer to global variable X:

view sc-X

procedure p1, p2

Select <p1, p2> **such that** IS-IN (<p1, p2>, structure-chart) **and**

IS-IN (p1, ref-X) **and** IS-IN (p2, ref-X)

To attach data interface information to procedure calling relationships, we might start with the following simplified view:

view first-cut-proc-interface

Select <p1, p2, globVar> **such that** IS-IN (<p1, p2>, structure-chart) **and**

(Modifies (p1, globVar) **and** Uses (p2, globVar)) **or**

(Uses (p1, globVar) **and** Modifies (p2, globVar))

To define a view that better approximates procedure interfaces, we need to further constrain the above view addressing reachability of data definitions. We do this in the following way:

view pass-value-to

procedure p1, p2; statement s

Select <p1, p2, globVar> **such that** IS-IN (<p1, p2, globVar>, first-cut-sc-interface)

such that Parent* (p1, s) **and** Modifies (s, globVar) **and** Next* (s, callProc)

with callProc.procName=p2.procName

view sc-interface

procedure p1, p2; statement s, first

Select <p1, p2, globVar> **such that** IS-IN (<p1, p2, globVar>, pass-value-to)

such that Parent* (p2, s) **and** Uses (s, globVar) **and** First (first, p2)

and Affects (first, s, globVar)

The last view represents a structure chart together with interface information. Many reverse engineering views are best presented to a programmer in graphical form. We wrote a reverse engineering front-end for a commercial CASE tool in *PQL*. Program views obtained with *PQL* are translated into the format of a CASE import facility, loaded into the CASE repository and displayed using CASE editors.

More general views can be obtained through parameterization. For example, view 'use-X' is an instance of the following parameterized view:

view use-global-var (STR name)

Select procedure **such that** Uses (procedure, globVar) **with** globVar.varName=name

V. Evaluation of *PQL*

We tested expressive power of *PQL* in the following ways:

1. We compared *PQL* to program query/pattern notations developed in other research projects [1,4,7,12,14,16]. The comparison is favorable for *PQL*. We conveniently expressed all types of queries we found in those sources.
2. We studied program views supported by two commercial program analysis tools [19,21]. Again, we could formalize these views in *PQL*.
3. We specified reverse engineering transformations in *PQL*. The source language was COBOL-85 with data stored in flat files. We described structure chart views (see examples in section IV E) and data model views. We recovered the first-cut Entity-Relationship (ER) data model from flat file structures based on heuristic rules such as:
    - candidate entities correspond to files,
    - certain sub-records and repeated groups are also candidates for entities,
    - record fields are candidates for entity attributes,
    - embedded foreign keys indicate entity relationships.

    Reverse engineering is done in steps, with user being involved at the end of each step. Views to be obtained at each step are described in *PQL*. The user (data analyst) accepts/rejects selection made automatically by the tool, before reverse engineering progresses to the next, higher-level, step. In this experiment, *PQL* proved to be effective in specifying reverse engineering transformations.

VI. PKB implementations

Program Knowledge Bases (PKB) can be implemented on a variety of media such as general-purpose relational database [4,14], special-purpose relational database [13], object-oriented database [1,12], expert system [6], abstract syntax trees [7,16], dependency graph [12,15] and a hybrid PKB [9,10,11]. We experiment with PROLOG and hybrid implementations of the PKB. Translation of the program model into PROLOG schema is straightforward.

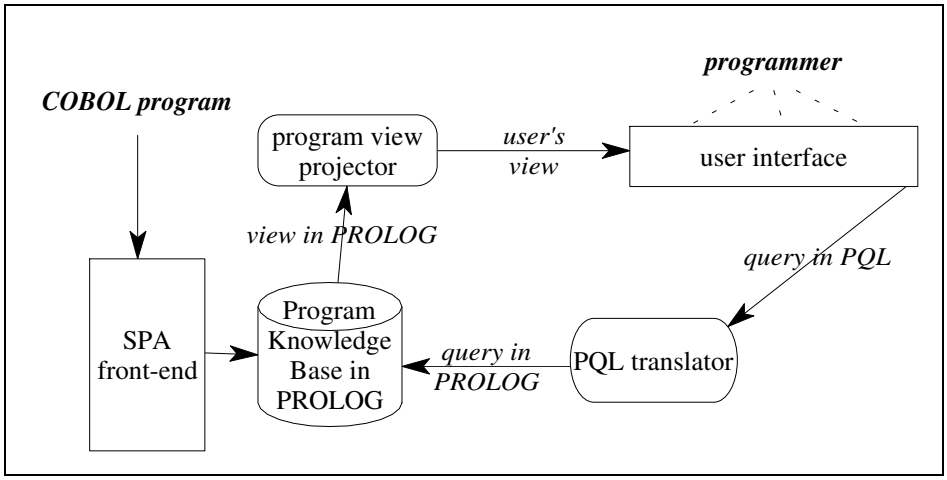

Fig. 5 PROLOG implementation of *PQL*

The hybrid PKB combines attribute syntax trees with PROLOG [9,10,11]. Global design program model is stored in the PROLOG database, while program structure and detail program design models are stored as attribute syntax trees. Both program representations are tightly integrated, so that program entities and relationships stored in a PROLOG database can be traced to syntax trees and vice versa. *PQL* queries can be efficiently resolved on the hybrid PKB. The concept of a hybrid PKB for storing programs was developed for language-based software development environments [9]. Our experiment shows that a hybrid PKB serves well needs of SPAs.

VII. Conclusions

We designed Program Query Language (*PQL*) as a general-purpose query language for Static Program Analyzers (SPA). *PQL* has the following features:

1. *High expressive power and simplicity.* In *PQL* we can query on system-level design as well as to search for constrained code patterns. A programmer writes *PQL* queries in terms of an explicit conceptual program model.
2. *Hierarchical queries*. Complex program views can be expressed in terms of simpler ones. This feature makes it possible to formulate queries through levels of abstraction leading from code structures up to the level of program conceptualizations.
3. *Language-independence of PQL.* Same program modeling conventions are used independently of a source language. *PQL* queries are driven by the program model.

PQL can be used as an end-user language. However, formulation of complex queries directly in *PQL* may not be easy. Therefore, end-user interfaces to help programmers in query formulation should be built upon *PQL.* In future work, we plan to further refine *PQL* with

new features (such as parameterized views). The long-term goal of the work described in this paper is to define a framework for systematic design of static program analyzers. The purpose of this framework is to study tool capabilities in context of the maintenance processes and programmer's behavior. The premise of our approach is that models of tool environment, i.e., of an underlying software process and programmer behavior, should be explicitly formulated and thoroughly analyzed as an integral part of the tool development cycle. We believe that by doing so we can build better tools and that, in long-term, such a systematic design will have a positive impact on tool flexibility and production cost [10]. Although we experiment with maintenance processes and SPA tools, we feel that the same design concepts also apply to other software processes and other types of tools.

Acknowledgments

This work is supported by National University of Singapore Research Grant RP920613. Chan's work on the ER query language [2] influenced the design of *PQL*. Thanks are due to Shen Han for insightful comments. The following NUS students implement *PQL* and various components of the SPA generation system: Lu Chay Woon (SPA for COBOL-85), Lee Hwee Ling (*PQL* on PROLOG PKB), Wang Guosheng (*PQL* on a hybrid PKB) and Tan Poh Keam (reverse engineering of data into a CASE tool).

References

[1] Burson, S., Kotik, G. and Markosian, L. "A Program Transformation Approach to Automating Software Re-enginering," *Proc. COMPSAC'90*, pp. 314-322

[2] Chan, H. C. "An Entity-Relationship Enhanced Logic System," *Proc. Second Int. Symp. on Database Systems for Advanced Applications*, Tokyo, 1991, pp. 401-410

[3] Chen, P. "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Database Systems, 1*, 1 (1976), 9-36

[4] Chen, Y., Nishimito, M. and Ramamoorthy, C. "C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, March 1990, pp. 325-334

[5] Date, C. and Darwen, H. *A Guide to the SQL Standard*, Addison-Wesley, 1993

[6] Devanbu, P. Brachman, R.J., Selfridge, P.G. and Ballard, B.W. "LaSSIE: A Knowledge-Based Software Information System," *CACM*, vol. 34, No. 5, May 1991, pp. 34-49

[7] Devanbu, P. "GENOA - A Customizable, Language- and Front-end independent Code Analyzer," *Proc. 14th Int. Conf. on Software Engineering*, 1992, pp. 307-319

[8] Horwitz, S., and Teitelbaum, T. "Relations and Attributes: A Symbiotic Basis for Editing Environments," *Proc. ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle, June 1985, pp. 93-106

[9] Jarzabek, S. "Specifying and Generating Multilanguage Software Development Environments," *Software Engineering Journal*, vol. 5, no. 2, March 1990, pp. 125-137

[10] Jarzabek, S. “Systematic Design of Static Program Analyzers,” *18th Computer Software and Applications Conference*, *COMPSAC’94*, Taipei, November 1994, pp. 281-286

[11] Jarzabek, S., Shen, H. and Chan, H.C. “A hybrid Program Knowledge Base system for Static Program Analyzers,” accepted for presentation at Asia Pacific Software Engineering Conference, APSEC’94, Tokyo, December 1994

[12] Kozaczynski, W., Ning, J. and Engberts, A. “Program Concept Recognition and Transformation,” *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992, pp. 1065-1075

[13] Lewerentz, C. "Extended Programming in the Large in a Software Development Environment," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, November 1988, pp. 173-182

[14] Linton, M.A. "Implementing Relational Views of Programs," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 65-72

[15] Ottenstein, K. and Ottenstein, L. "The Program Dependence Graph in a Software Development Environemnt," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 177-184

[16] Paul, S. and Prakash, A. “A Framework for Source Code Search Using Program Patterns,” IEEE Transactions on Software Engineering, vol. 20, no. 6, June 1994, pp. 463-474

[17] Reps, T.W., and Teitelbaum, T. "The Synthesizer Generator," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 42-48

[18] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

[19] Seet, C. “Analysing and Reverse Engineering COBOL Programs,” *Proc. Sigapore Computer Society Silver Jubilee Conference on Software Enginering*, Singapore, October 1992, pp. 175-188

[20] Stonebraker, M., Kreps, P. and Held, G. "The Design and Implementation of INGRES," *ACM Transactions on Database Systems 1* (3), 1976

[21] VIA/CENTER, *VIASOFT*

[22] Weiser M. "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, July 1984, pp. 352-357