PQTL: a langauge for specifying program transformations

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

Abstract

We describe a program query and transformation language, *PQTL* for short, a general-purpose, end-user level language to specify program transformations. *PQTL* provides a basis for automating program transformations in software evolution, conversion and re-engineering. In *PQTL*, we specify program transformations in terms of the logical program design model. Separation of the logical program design model from the actual mechanism used to compute and store program designs allows us to design a generic program transformation tool. The generic tool can be customized to the needs of a specific program transformation project, i.e., to the source language,to specific program transformation rules and to an internal program representation. *PQTL* is an extension of *PQL*, a program query language, that we use to specify program queries in the interactive analysis of programs for understanding.

1. Introduction

Two types of program transformations are needed in software evolution. ***Code-to-code program transformations*** restructure code to improve program readability or re-engineer programs to new software/hardware architectures. In software re-engineering [4], which is the most complex code-to-code program transformation task, program design itself is modified and the program code is re-generated according to the new design plan. Examples of software re-engineering projects include migration from COBOL flat files to a relational database, re-designing programs into the client-server architecture or changing the implementation technique (e.g., re-designing a procedural program into the Object-Oriented architecture or migrating COBOL programs into a CASE tool). ***Reverse engineering program transformations***, extract design abstractions from code for the purpose of program understanding, modification or re-engineering. We observe that in any but trivial program

transformation task, we need some program design information to guide the program transformation process.

Generally, the more sophisticated program transformation task, the higher level program design information is needed to accomplish the task. Fig. 1 depicts program transformation tasks of various degrees of complexity. A PKB (Program Knowledge Base) is a repository of **low level program design abstractions** such as program syntax trees attributed with semantic information, control flow graphs, data flow relations, procedure call graphs, etc. The *Refronte* (Fig. 1) is a *r*everse *e*ngineering *front-e*nd that extracts low level design abstractions from source programs and stores them in the PKB.

The PKB assists in recovering **high level design abstractions** (stored in Design Knowledge Bases, DKB-*i* in Fig. 1). These design abstractions are recovered by applying heuristic rules, by filtering lower level design abstractions and with the additional input from a domain expert. An example of a **heuristic rule** is mapping of files to entities and foreign keys to entity relationships in recovering Entity-Relationship (ER) data models [2] from file structure definitions [10]. An example of a **reverse engineering filter** is slicing a possibly huge procedure call graph, based on the analysis of data coupling between procedures [10] (the objective here may be to identify procedures that should be packaged into a module). An example of an **input from a domain expert** is the selection of meaningful names for data during data restructuring or deciding which files represent well defined application domain concepts and should be mapped to entities in the ER data model. As indicated in Fig. 1, the design abstractions stored in the PKB and DKBs assist in program transformations.

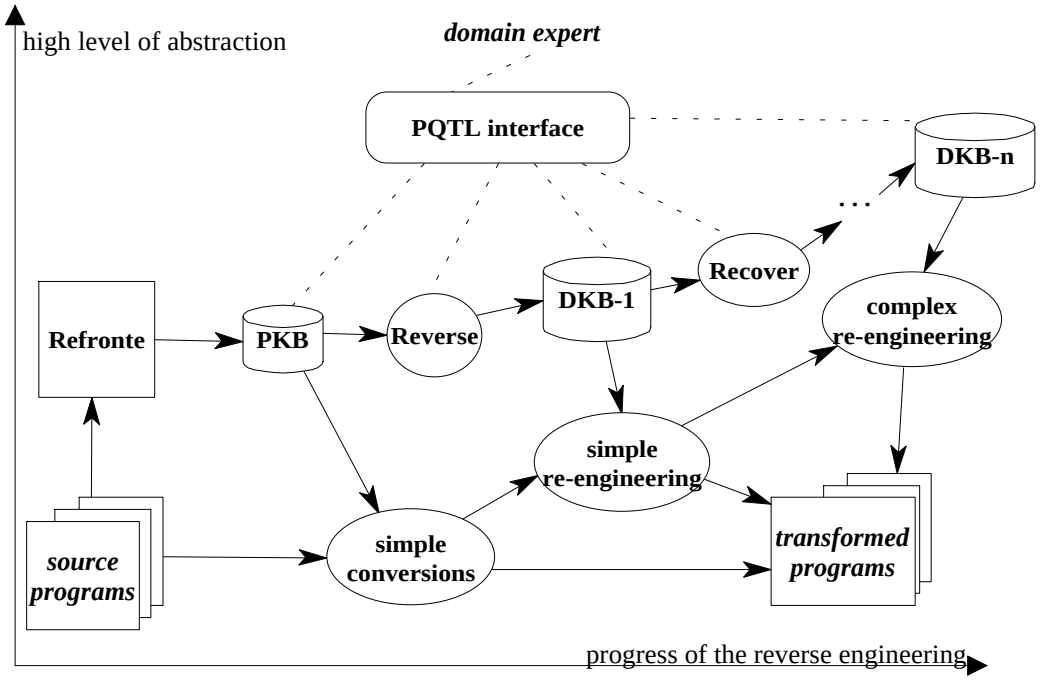

Fig. 1. Types of program transformation projects

We developed a notation, called *p*rogram *q*uery *l*anguage (*PQL* for short), for specifying program patterns and program queries [7,8,9]. *PQL* provides a user interface for static program analysis tools that help in program understanding by answering programmer's queries about programs. In this paper, we describe an extended *PQL*, called *p*rogram *q*uery and *t*ransformation *l*anguage (*PQTL* for short) to specify program transformations. *PQTL* provides a basis for building tools to assist in program transformation projects. The main contributions of the *PQTL* are its simplicity and generality. We separated the program transformation mechanism from the actual mechanism used to compute and store the program design information. We express program transformations in terms of the logical program design model. Tools built on the *PQTL* are adaptable to different source languages, can work with different program design representation media and provide a flexible mechanism to define program transformation rules.

Most program conversion and re-engineering tasks cannot be accomplished in one shot. Therefore, we assume an incremental program transformation process, with a domain expert participating in the process and providing additional inputs [10].

The issue of computing low level design abstractions and storing them in the PKB by no means is a trivial one. However, the physical media for the PKB will not be addressed in this paper. Many authors [1,5,6,7,11,13] described, evaluated and compared various media to represent program design, such as relational databases, special-purpose databases, Object-

Oriented data bases, PROLOG, attribute syntax tress, program dependence graphs and hybrid media that integrate relational databases (to store global program designs) with attribute syntax trees (to store detailed program design information). We refer the reader to the above sources. *In this paper, we shall describe a method for specifying program transformations in terms of logical program models rather than in terms of their physical representation.* The reason why we address program transformations at the program design representations-independent level is that we are looking for generic solutions to program transformation problems. In [9], we described how program design queries can be addressed in the language-independent way and in [10] we described the design of a generic reverse engineering tool. In development of database applications, we discovered long time ago the benefits of separating the physical data representation from the logical data models. By doing this, we can write flexible database applications independently of the physical storage for data. *The aim of this paper is to present a program transformation language that is based on the logical program design model. We show how this language facilitates the design of a generic and flexible program transformation system.*

The paper is organized as follows: in the next section we classify program transformations. Then, we describe *PQTL* and examples of program transformations in *PQTL*.

2. Classification of program transformations

Program transformations involve program code and a program design model. *Code-to-code* transformations have the following formats:

(1) $ccTrivial : \text{Code} \rightarrow \text{Code}$

or

(1a) $ccTransf : \text{Code} \times \text{PKB} \rightarrow \text{Code} \times \text{PKB}$

or

(1b) $ccTransf : \text{Code} \times \text{PKB} \rightarrow \text{Code} \times \text{PKB} \times \text{DKB}$

or

(1c) $ccTransf : \text{Code} \times \text{PKB} \times \text{DKB} \rightarrow \text{Code} \times \text{PKB}$

or

(1d) $ccTransf : \text{Code} \times \text{PKB} \times \text{DKB} \rightarrow \text{Code} \times \text{PKB} \times \text{DKB}$

We observe that even simple transformations such as renaming of variable X to Y in a program require some knowledge of program design. In the case of renaming variables, we must distinguish occurrences of X as an identifier from other occurrences of X in the program

text. We may also need to take into the account program scoping rules. Code-to-code transformations may involve low level design abstractions (PKB) and/or high level design abstractions (DKB). Transformations may affect PKB, DKB or both. Variants 1a -1d of code-to-code transformations reflect these situations. Transformations 1a and 1b usually correspond to simple program conversions, while transformations 1c and 1d reflect software re-engineering projects.

Reverse engineering (or: *code-to-design*) transformations have one of the following formats:

(2a) *revTransf*: Code × PKB → PKB

or

(2b) *revTransf* : Code × PKB → PKB × DKB

or

(2c) *revTransf* : Code × PKB × DKB → PKB × DKB

Reverse engineering transformations do not modify design abstractions already stored in the PKB and DKB. Reverse engineering either filters information to provide a selective, focused design view or creates new design abstractions. If required, we should be able to store filtered design views and new design abstractions in the PKB and DKB for future use.

We formally specify program transformations by:

a) defining logical program design models (i.e., logical models for the PKB and DKBs),
b) defining a formal notation to specify a class of code and design patterns, and
c) defining a notation to specify mappings between patterns.

3. Specifying program transformations in *PQTL*

Program design modeling conventions and pattern specifications in *PQTL* are the same as in *PQL*. We refer the reader to [7,8,9] for the details, but for the convenience, we included a summary of program modeling conventions and program design pattern specifications in the appendix. In examples below, we refer to program models described in the appendix.

3.1 Code-to-code program transformations

Code-to-code transformations modify specified code patterns:

source-pattern **ReplaceWith** target-pattern

Example 1. Renaming variables

Suppose we wish to rename all the references to a global variable X with Y in the statement part of all the program procedures.

```
globVar vX, vY
Select vX  with vx.varName = "X" from  stmtPart
        ReplaceWith vY with vY.varName = "Y"
```

Explanation: The **Select** clause specifies a source pattern. The **ReplaceWith** clause specifies the target pattern that is to replace the source pattern. Notice that in transformation specification, we treat a variable as a meaningful program entity - see program models of Fig. 2, 3 and 4a in appendix - not just a string of characters. As a result, the transformation does not put the syntax tree into an inconsistent state.

Example 2. Suppose we want to change "a+b" to "a*b*c" in all the expressions that appear as loop or if-goto conditions.

```
var va, vb, vc
Select plus (va, vb) with va.varName="a" and vb.varName="b"
  where (while-do (expr, _) or if-goto (expr)) such that Contains* (expr, plus)
        ReplaceWith  times (va, times(vb, vc)) with vc.varName="c"
```

Example 3. Suppose now that we want to perform the above transformation only for those expressions that are affected by assignments to variable "a" that appears within a loop.

```
var va, vb, vc
expr e1, e2
Select plus (va, vb) with va.varName="a" and vb.varName="b"
  where (while-do (e1, _) or if-goto (e1)) such that Parent* (e1, plus) and
  exists [assign where assign (va, e2) such that Parent* (e2, va) and Affects* (assign, e1)]
        ReplaceWith  times (va, times(vb, vc)) with vc.varName="c"
```

3.2 Specifying reverse engineering filters in *PQTL*

In this section, we show how we specify reverse engineering filters in *PQTL*. *PQTL* allows us to build complex patterns out of already constructed, simpler ones. With hierarchical patterns, one can systematically progress from low level code patterns to higher levels design patterns. Each program entity denotes a set of its instances. Binary relationship is treated a set of the pairs of interrelated entity instances. Patterns are typed (Table 1). Built-in operations (such as CARD) apply to sets and tuples (Table 2).

pattern	**pattern type**
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
Select assign*	lists of assignments

Table 1. Examples of pattern types

Operations on sets and lists	**the meaning**
CARD (s)	the number of elements in s
IS-IN (e, s)	TRUE, if e is in s; otherwise, FALSE
NOT-IN (e, s)	TRUE, if e is not in s; otherwise, FALSE
Other operations on lists	
SUB-SEQ (t1, t2)	TRUE, if t1 appears in t2
FIRST (s, t), LAST(s, t)	TRUE, if s is first (last) element of list t

Table 2. Operations on sets and sequences

The result of pattern matching is called a view. Views can be named. Views retrieved in one **Select** clause can be used in specifications of other patterns. Hierarchical patterns and views can be constructed in that way. Here is an example:

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

A procedure call graph (also called a structure chart diagram) for the whole system can be defined as:

view pcg
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. So we might want to compute a slice of the procedure call graph showing, for example, only those procedures that communicate through a global variable X. We assume that the *Refronte*

builds a procedure call graph, computes data usage information, etc. based on the analysis of a source program. In the PKB, the procedure call graph is represented by the many-to-many relationship Calls (procedure, procedure). Relationships Modifies and Uses record data usage information. We can specify the first-cut slice of the procedure call graph pcq-X as follows:

view pcq-X
procedure p1, p2
Select <p1, p2> **such that** Calls (p1, p2) **and** IS-IN (p1, mod-X) **and** IS-IN (p2, use-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>, pcg) **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-proc-interface) **and**
Parent* (p1, s) **and** Modifies (s, globVar) **and** Next* (s, callProc)
with callProc.procName=p2.procName
view pcg-interface
procedure p1, p2
statement s, first
Select <p1, p2, globVar> **such that** IS-IN (<p1, p2, globVar>, pass-value-to)
and Parent* (p2, s) **and** Uses (s, globVar) **and** First (first, p2) **and** Affects (first, s, globVar)

The last view represents a procedure call graph together with the interface information.

4. Specifying data reverse engineering in *PQTL*: an example

Suppose our objective is to recover the Entity-Relationship data model from COBOL 85 file definition structures. Reverse engineering of data is an essential step in re-engineering old data stores (such as flat files, hierarchical databases, etc.) into newer database architectures (such as relational or object-oriented databases).

Reverse engineering must separate the essential program design information from the implementation details. The DATA DIVISION is a good place to begin search for data design information. Program files are likely candidates for entities as they contain relatively static data. The record layouts in the FILE SECTION provide information about entity attributes. Another source of data design information is the FILE-CONTROL paragraph. It provides information about the file organization, the file records and file access keys. Other sections of a COBOL-85 program can be assumed to contain implementation-specific information. The

Refronte of our data reverse engineering tool produces descriptions of program data according to schema depicted in Fig. 2 and 3.

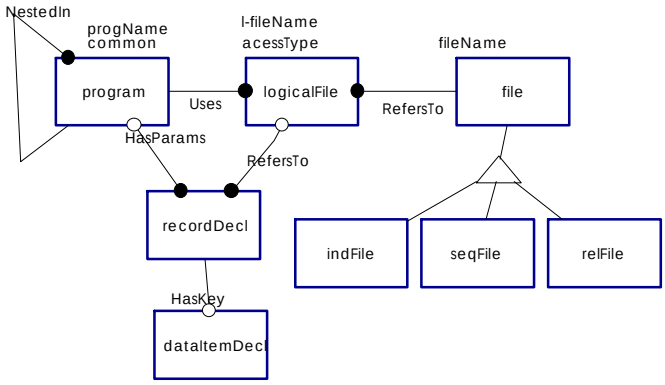

Fig. 2. Logical model of file descriptions

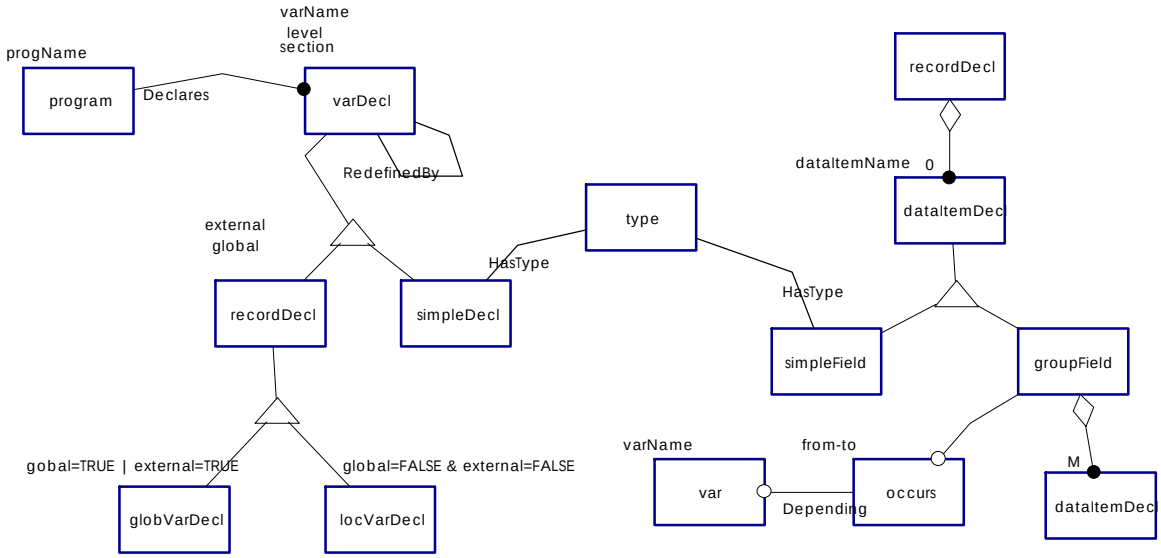

Fig. 3. Logical model of data descriptions

From Fig. 2 we read that a COBOL-85 program may use many files. While there can be only one physical name for a file in the operating system, a physical file can be referred to using different logical names in programs. Entities *logicalFile* and *file* model this situation. Links between logical files and physical files are defined within a program, in ASSIGN clause of FILE-CONTROL paragraph (sometimes links are established in JCL statements and interactive program set up procedures). All these sources are used to establish relationship *RefersTo* between entities *logicFile* and *file*. Files can be organized into three categories: sequential, indexed, and relative. This is modeled by *IsA* classification, represented by a triangle in diagrams. Indexed files have a corresponding key that is used to selectively read records from the files.

Diagrams of Fig. 3 define the structure of syntax entities such as *recordDecl* and *dataItemDecl*. Syntax entities represent language constructs and correspond to abstract syntax grammar symbols. Syntax entities are also classified using *IsA* relationship (represented by a triangle with a general entity above and specialized entities below). Attributes (displayed above the syntax entity box) assigned to a parent apply to all its children. Attributes specify types of information that are available at syntax tree nodes.

Sequences of elements are identified by aggregation 1-to-many relationship link. There are two sequences in Fig. 3, namely a sequence of data items in a record declaration (connectivity 0 above the *dataItemDecl* box indicates that this sequence may be empty) and a sequence data items in a sub-record (groupField). Syntax entity *occurs* is an optional component of a *groupField*. Entities in structure diagrams can be given roles, e.g., *Depending* is the role of entity *var* as a component of construct *occurs*.

The following heuristics can be used to generate the first-cut ER data model from file structure descriptions:

1. files are candidates for entities,
2. record fields are candidates for attributes for the entities,
3. sub-records including more than one field are candidates for entities,
4. repeated sub-records (OCCURS) including more than one field are candidates for entities,
5. foreign keys indicate entity relationships,
6. one-to-many relationship occurs between an entity containing a repeated group of record fields and the entity that represents this group,
7. primary key for a file is also the identifier for its corresponding entity.

Below are two view definitions that correspond to the data reverse engineering heuristics:

R1. Extract candidate entities for the ER data model

file, dataItemDecl *IsA* ERentity

view file-ent

Select file

view group-ent

dataItemDecl+ dataItems

Select groupField **such that** Parent* (recordDecl, groupField) **and**recordDecl.section=File-Section

and groupField(dataItems) **and** LENGTH (dataItems) > 1

view entities

Select ERentity **such that** IS-IN (ERentity, file-ent) **or** IS-IN (ERentity, group-ent)

Explanations: above, we have three view definitions. Each view is named. A set of candidate entities is defined in the last view as a union of views *file-ent* and *group-ent*. The first line introduces variable *ERentity* as a super-class of *file* and *dataItemDecl* (*IsA* classifications has usual meaning). In the second view, we introduce *dataItems* as a synonym for a sequence of *dataItemDecl*s. Then, we select all sub-records of length more than 1 from records declared in the file section of programs under consideration.

R2. Find record fields that possibly are foreign keys

view foreign-keys

simpleField recField, foreignKey

Select foreignKey **such that** Parent* (recordDecl, foreignKey) **with** recordDecl.section = File-Section

such that exists [recField **such that** foreignKey.dataItemName = recField.dataItemName

and exists [recordDecl **such that** HasKey (RecordDecl, recField)]]

Explanations: the second line declares two synonyms for *simpleField*, namely *recField* and *foreignKey*. Then, we consider records declared in file sections (those are the records that possibly correspond to ER entities) and select those record field whose names are identical to other record fields that are known to be keys. We do not consider composite keys in this example.

Candidate entities from the view obtained by applying rule R1 are presented to a domain expert. A domain expert can further explore the PKB to determine which files represent true domain concepts and should be selected as entities in the ER data model. Similarly, a domain expert would analyze information selected in foreign-key view to exclude accidentally identical record field names as possible foreign keys. True foreign keys would then be interpreted as relationships between entities. After these steps, the first-cut ER data model can be generated.

5. Mappings across different program design models

In the above examples, we assumed that the design models for the source and target programs are the same. In general, this assumption may be too restrictive. Indeed, we may need to define separate models for the source PKB and DKB and for the target PKB and DKB. As an example, consider conversion of FORTRAN programs to ADA. Many of the

programming concepts from ADA (such as package or module specifications) are not present in FORTRAN. Even much simpler conversions from COBOL 74 to COBOL 85 involve similar problems. To formalize transformations across dissimilar language domains, we need to refine our transformation schema described in section 2 in the following way:

(1a) $ccTrivial : \text{Code} \rightarrow \text{Code}$

or

(1b) $ccTransf : \text{Code} \times \text{PKB} \rightarrow \text{Code'} \times \text{PKB'}$

or

(1b) $ccTransf : \text{Code} \times \text{PKB} \rightarrow \text{Code'} \times \text{PKB'} \times \text{DKB'}$

or

(1c) $ccTransf : \text{Code} \times \text{PKB} \times \text{DKB} \rightarrow \text{Code'} \times \text{PKB'}$

or

(1d) $ccTransf : \text{Code} \times \text{PKB} \times \text{DKB} \rightarrow \text{Code'} \times \text{PKB'} \times \text{DKB'}$

where models with primes refer to the target language.

As an example, consider re-engineering of C programs into C++. Suppose we wish to identify candidate classes in a C program. To achieve this, we might want to formalize (among others) a transformation rule that says that every function that uses a certain data structure, say data-X, should be considered as a candidate method for class X. In the reality, the process that leads to identifying class X may involve applying heuristics (such as data coupling between C functions) and manual analysis of programs by a domain expert. As notions of a class, methods and data representation for objects are not present in the domain of C programs, we need to model C++ programs independently of C programs. Simplified program models that we need to express required transformations are depicted in Fig. 4.

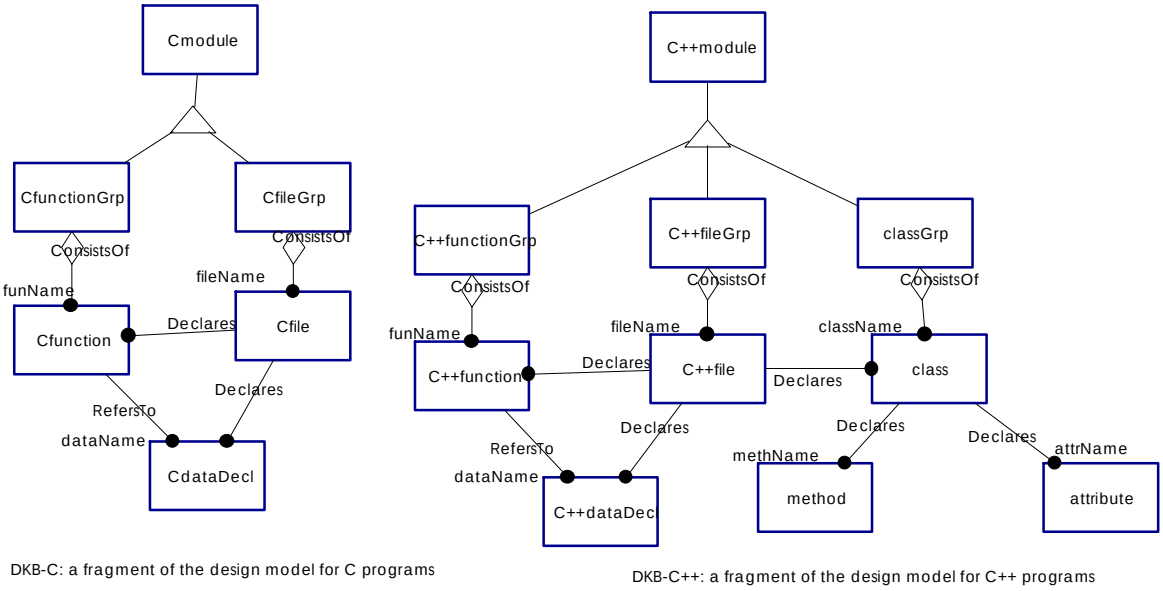

Fig. 4. Design models in C to C++ program re-engineering

Mappings across different design models have the following format:

source-pattern [**CreateEnt** designEntitySpec]

[**CreateRel** relSpec]

[**CreateLink** linkedDesignEntities]

In our example, the source-pattern refers to the C program design model (DKB-C), while the right hand side design entities and relationships refer to the C++ program design model (DKB-C++). Operator **CreateEnt** creates a new instance of a design entity, **CreateRel** establishes a relationship between two design entities and **CreateLink** links design entities from different program design models for the purpose of traceability.

In transforming a C program design into the C++ design, we shall apply the following simple heuristics:

H1. If a data structure, say data-X, is used in more than two C functions, then we consider candidate class X with data-X being data representation for class X.

H2. Functions that refer to data-X are considered candidate methods for class X.

With reference to the above program models and heuristics, we now outline a semi-automatic process to identify candidate classes in C code. For each processing step, we state whether the task is carried out by a domain expert (DE), a programmer (PR) or an automatic *PQTL*-based program transformation tool (*PQTL*).

1. DE: Analyze a program application domain and identify candidate classes in the application domain.

2. DE-PR-*PQTL*: Identify data structures in C programs that may be relevant to identified classes.

   This step involves a domain expert, programmer and the automatic program transformation system. We apply the heuristic rule H1 to find data structures that may be data representations for a class. The following *PQTL* transformation reflects the heuristic H1:

   view dataRep

   Select CdataDecl **such that** RefersTo (Cfunction, CdataDecl>2)

   Selected data structures are possible manifestations of classes in a C program. A programmer and domain expert must validate automatically identified data.

3. *PQTL*: For a given data structure in set dataRep, say data-X, select files that refer to data-X.

   view files-X

   Select Cfile **such that** RefersTo (Cfile, data-X)

4. DE-PR-*PQTL*: Apply heuristic rule H2 to find candidate methods for class X

   view methods-X

   Select Cfunction **from** files-X **such that** ReferesTo (Cfunction, data-X)

   As before, a domain expert and programmer must review and validate automatically selected candidates for methods.

5. *PQTL*: Based on our findings, we can now define *PQTL* transformations to create the first-cut C++ program design.

   5a. Create classes in the C++ DKB.

   For each data in set dataRep, we create a class with the same name (operator **CreateEnt**):

   CdataDecl data

   Select data **from** dataRep

   CreateEnt class **with** class.className=data.dataName

   5b. Create methods for class X in the DKB-C++. Relate methods to the corresponding class (operator **CreateRel**) and link functions (operator **CreateLink**) to corresponding methods for the purpose of traceability:

   Cfunction fun

   Select fun **from** methods-X

CreateEnt method **with** method.methName=fun.funName

CreateRel Declares(class, method) **with** class.className="X"

CreateLink (fun, method)

5c. Create attributes for class X in the DKB-C++.

CdataDecl data

Select data **from** dataRep

CreateEnt attribute **with** attribute.attrName=data.dataName

CreateRel Declares(class, attribute) **with** class.className="X"

CreateLink (data, attribute)

6. Conclusions

In this paper we described a program query and transformation language (*PQTL* for short). We designed the *PQTL* as an end-user language to specify program transformations in software conversion and re-engineering projects. Pattern specification mechanism in *PQTL* is the same as in *PQL* [7,8,9], an end-user language to query programs during maintenance. The *PQTL* adds program transformation capability to the *PQL*.

Distinguishing features of the *PQTL* are simplicity, independence of the source language and independence of the actual media used to store program designs. We use the *PQTL* to design generic program transformation systems that can automate simple conversions and assist programmers in complex software re-engineering projects.

A critical decision in designing the *PQTL* was to separate the logical program design schema from the physical representation of program designs stored in the PKB and DKBs. We use an extended OMT notation to model the logical structure of the required program design information. The physical PKB may be implemented on a variety of media including PROLOG (very useful in prototyping), a relational database, an Object-Oriented database or as an attribute syntax tree. Practical systems that must deal with big sources can use a hybrid representation [6,7] that combines attribute syntax trees (to store the detailed program information) and a relational database (to store the global design information). In the interactive program transformation situation, it may be more appropriate to pre-compute only essential program designs (such as syntax trees, control flow graphs and procedure call graphs) and compute other types of detailed information (such as data flow relations) on demand. All these are very important, implementation-dependent decisions. The logical model of program information makes it possible to define a generic core of a program transformation

tool independently of those decisions, independently of the actual mechanism used to compute and store program designs.

Based on the above arrangement, we design a generic program transformation tool that can be customized to the needs of a re-engineering project in hand. During customization of the program transformation tool to a specific program transformation project, we proceed as follows:

1. Start by creating logical program design models for the PKB and DKBs.
2. Express the required program transformations in *PQTL* in terms of the logical models.
3. Decide upon the physical structure of the PKB and DKBs.
4. Use a compiler generator to implement the *Refronte* to parse source programs. Write the PKB generation actions in terms of logical models, addressing only the program design information to be permanently stored in the PKB.
5. Implement an interpreter of logical program models in terms of the physical representation of the PKB and DKBs.
6. Generate the *PQTL* translator to convert program transformations defined in terms of logical models into the equivalent transformations on the physical representation of the PKB and DKBs.

We have implemented program queries, view definition and reverse engineering part of the *PQTL*. We are still experimenting with specifying code-to-code and cross domain transformations. This part of the system have not been implemented yet. In future work, we shall try to identify and formalize possible wide range of program transformation heuristics to refine the *PQTL* specification features. We shall experiment with various source/target languages and various media to store program designs. The ultimate goal of our work in the area of tools for software evolution is to understand the interplay between static program analysis, reverse engineering and program transformation methods. We believe powerful tool environments for software evolution should address the three methods in an integrated way. Form the tool designer perspective, our experience shows that the underlying design concepts and generic solutions are similar for the tree types of tools.

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

[3] 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
[4] Chikofsky, E.J. and Cross, J.H. (1990). "Reverse Engineering and Design Recovery: a Taxonomy," *IEEE Software*, Vol. 7, No 1, January 1990, pp. 13-18
[5] Devanbu, P. "GENOA - A Customizable, Language- and Front-end independent Code Analyzer," *Proc. 14th Int. Conf. on Software Engineering*, 1992, pp. 307-319
[6] Jarzabek, S. "Specifying and Generating Multilanguage Software Development Environments," *Software Engineering Journal*, March 1990, pp. 125-137
[7] Jarzabek, S., Shen, H. and Chan, H.C. "A Hybrid Program Knowledge Base for Static Program Analyzers," *Proc. Asia-Pacific Software Engineering Conf., APSEC'94*, Tokyo, Dec. 1994, pp. 400-409
[8] Jarzabek, S. "Systematic Design of Static Program Analyzers," *Proc. COMPSAC'94*, IEEE Comp. Soc. Press, Taipei, November 1994, pp. 281-286
[9] Jarzabek, S. "*PQL*: A Language for Specifying Abstract Program Views," *Technical Report No. TR11/94*, DISCS, National University of Singapore, November 1994; accepted for the *5th European Software Engineering Conference, ESEC'95*, Barcelona, September 1995
[10] Jarzabek, S and Tan, P.K. "Design of a generic reverse engineering assistant tool," accepted for *2nd Working Conference on Reverse Engineering, WCRE*, Toronto, Canada, July 14-16, 1995
[11] Linton, M.A. "Implementing Relational Views of Programs," *Proc. Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 65-72
[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] Ottenstein, K. and Ottenstein, L. "The Program Dependence Graph in a Software Development Environemnt," *Proc. Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 177-184
[14] 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
[15] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

Appendix. Program models

This appendix contains program models that are needed to understand examples of program patterns and program transformations in section 3. Code and design patterns are expressed in terms of explicit conceptual models of the PKB and DKB. The program modeling notation described here is the same as in case of *PQL* [7,8,9]. We use the notation of an extended OMT [15].

Modeling program design

Fig. 1 depicts an example of a global design model for programs in L. The global design model is expressed in terms of *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*' - its transitive closure: Calls* (p, q) **iff** Calls (p, p1) **and** Calls* (p1, q)

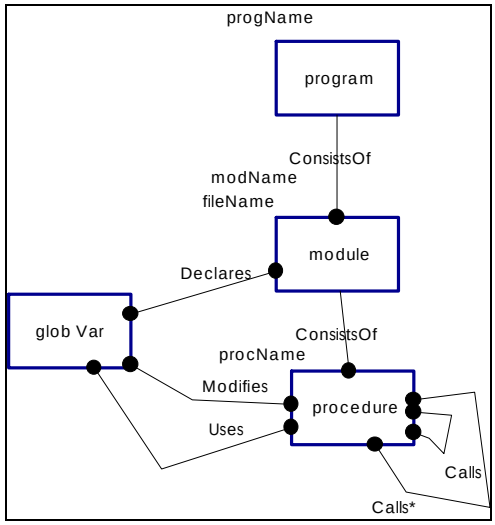

Fig. 1. An example of a global program design model

An example of the detail program design model is shown in Fig. 2.

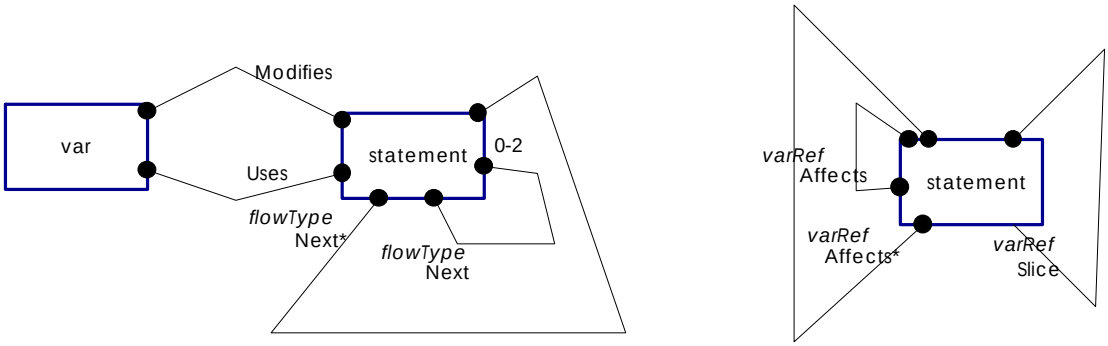

Fig. 2. An example of a detail program design model

Modeling program structure

Program structure is best modeled by an abstract syntax grammar and then represented by syntax trees. A grammar for L in graphical form is given in Fig. 3a.

The program structure model uses similar conventions to those used in program design models. Grammar symbols (in boxes) represent *syntax entities*. Syntax entities are classified using IsA relationship (a triangle with a general entity above and specialized entities below). Attributes (displayed above the syntax entity box) assigned to a parent apply to all its children. Attributes specify types of information that must be available at syntax tree nodes. Lists are identified by aggregation 1-to-many relationship link. There are two types of lists, namely a list of locally declared variables (connectivity 0 above the ‘var’ box indicates that this tuple may be empty) and a list of statements. Relationship ‘RefersTo’ indicates that each reference to a variable is liked to the corresponding declaration (e.g., an entry in the symbol table). Attribute ‘varName’ for variable reference is, therefore, redundant.

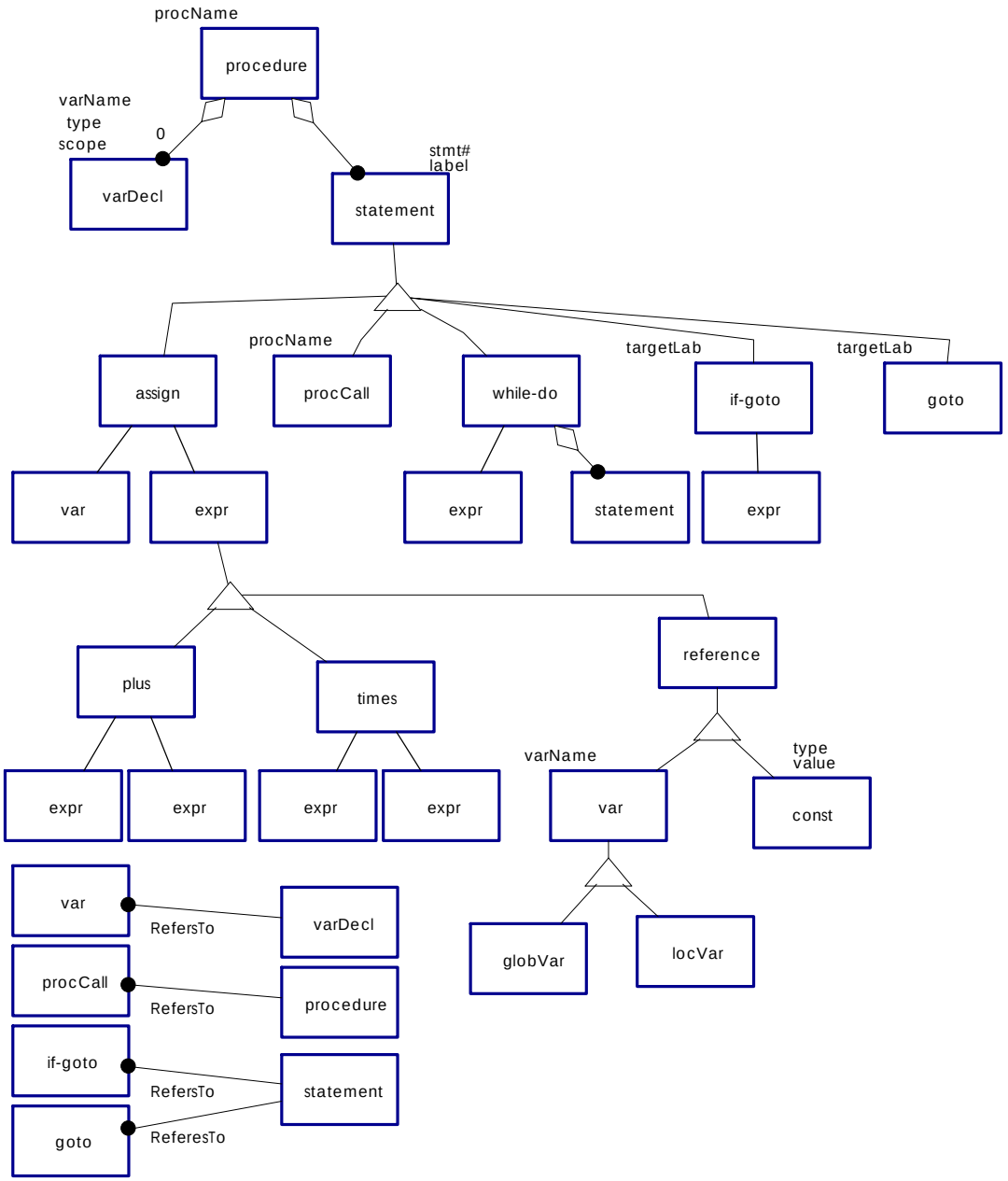

Fig. 3a A program structure model for language L

We extend the program structure model with abstract entities and relationships that are useful in formulating program transformations (Fig. 3b). 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 iff n1 is the direct parent node of n2 in a syntax tree. Parent* is a transitive closure of relationship Parent, i.e., Parent* (n1, n2) iff Parent (n1, x) and Parent* (x, n2). Similarly, relationship Follows (n1, n2) holds if n2 appears next to n1 in a list (e.g., in a list of statements). Follows* is a transitive closure of Follows, i.e., Follows* (n1, n2) iff Follows (n1, x) and Follows* (x, n2).

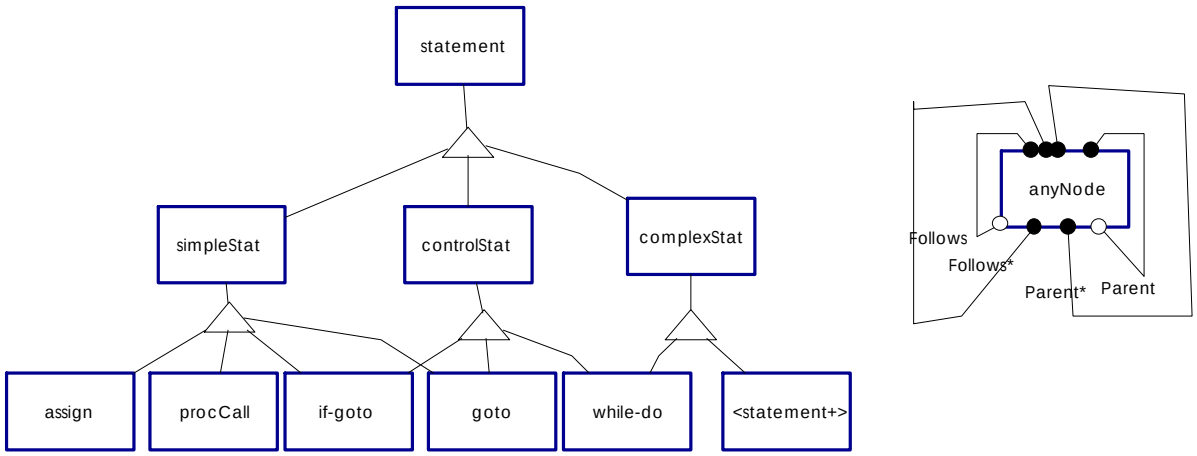

Fig 3b. Extensions to the basic program structure model

Box 1 shows the grammar rules in the predicate form. In *PQTL*, code patterns are expressed using predicate 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 1. Abstract syntax grammar for L