From an Object Semantic Model to a Design of SPA

Jin Song Dong Stan Jarzabek
School of Computing
National University of Singapore
dongjs,stan@comp.nus.edu.sg

Liam O' Brien
Mathematical Information Sciences
CSIRO, Australia
liam.obrien@cmis.csiro.au

Abstract

Static program analyzers are tools that enhance program understanding during maintenance by answering queries about programs. A number of program query languages have been proposed. The design of some static program analyzers has been based on these languages. However, few of these program query languages have a formal foundation. As a result, the interpretations of many program queries tend to be ambiguous, which often lead to an ambiguous design of static program analyzers. On the other hand, formal semantics of programming language are well studied, however, the traditional semantic models are unstructured and are difficult to reuse to provide a formal design for program analyzers. In this paper, we present an object-oriented model of the static semantics for a COBOL language subset. Based on this highly structured semantic model, a design of a static program analyzer including the models of program queries are formally presented.

1. Introduction

It is commonly known that a high percentage of programming resources are allocated to maintaining and reengineering existing code. There has been a growing interest in the software engineering community to develop tools that help software engineers to understand programs. Static program analyzers (SPAs) are tools that enhance program understanding during maintenance by answering queries about programs. SPAs can store source code information in a database, attribute syntax tree, dependency graphs, etc. Queries are evaluated based on that information.
A number of program query languages have been proposed [1, 8, 10]. The design of some static program analyzers has been based on these languages. However, few of these program query languages have a formal foundation. As a result, the interpretations of many program queries tend to be ambiguous, which often lead to an imprecise design of static program analyzers. Our earlier experience on the design and development an SPA based on PQL [8] has demonstrated that it is difficult to come up with a rigorous design of an SPA without defining the precise models of the relationships between program objects, e.g. the hierarchic relationships between programs and procedures, the relationship between procedures and their corresponding modified variables, the call graph, and etc. For example, in [8](Section 4), an instance of PQL to answer the query "Find programs that are called from more than 10 other programs" is

Select p (program) **such that** $calls(\_, p) > 10$

where the call graph $calls(\_, \_)$ has not been precisely defined in [8]. Perhaps, the call graph can be modeled as a relation between programs, i.e. $calls : Prog \leftrightarrow Prog$
Then the condition $calls(\_, p) > 10$ is still imprecise. More formally, in Z, this can be captured by the following predicate:

$$\#(calls \rhd \{p\}) > 10 \quad \text{or}$$
$$\#(\{p_1 : Prog \mid p_1 \ \underline{calls}\ p\}) > 10$$

The next question will be what is the precise definition of *calls*, that is, what condition makes two programs be related in the *calls* relation. To answer this question, we must pin down the semantics of program languages.
Formal semantics of programming languages often implicitly captures the program object relationships. However, the traditional semantic models are unstructured and are difficult to reuse to provide a formal design for program analyzers. Therefore, our approach is firstly to create a highly structured (object-oriented) semantic model for each of the target programming languages, then reuse the semantic model to capture the relationships between program objects, and finally present the formal design of an SPA by using those models. The formal design of SPAs will concentrate on modeling the precise relationship between the input and the expected output of each program query, without concern as to how to compute the query. This separation of concerns allows our formal model to be more generic and more abstract, so that it can be implemented in various ways.

To demonstrate our approach, we chose to design a COBOL SPA which is similar to the one suggested in [8]. We start by defining an object-oriented model of the static semantics for a COBOL language subset. Based on this, we formalise some important relationships between program objects that are useful in expressing a class of interesting program queries. Then, we present a formal design of a static program analyzer which includes the models of program queries[1].
We chose the Object-Z [6] notation as the meta-language to specify COBOL semantics and program queries because:

- The semantics of Object-Z itself is well studied. The denotational semantics [7] of Object-Z is closely related to the Z standard work).
- The object-oriented semantics of a program language is reusable and extendible. Object-Z has been effectively applied to programming language semantics [4].

It is assumed the reader is familiar with the basic Object-Z notation. Those less-basic aspects of Object-Z required for defining programming language semantics, such as class union, object containment (locality), etc. are discussed in Section 2. The static semantics of a COBOL language subset is defined in Section 3. Section 4 formalises some relationships between program objects in COBOL. Section 5 formalises program queries and presents a formal model of the SPA. Section 6 concludes the paper.

2. The Meta-notation: Object-Z Constructs

In this section, those specific aspects of Object-Z, such as class union, object locality, etc. needed in the paper to define the semantics of program queries are introduced and briefly discussed.

2.1. Class Union

Underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity that distinguishes it from every other object, regardless of class. If a file name is uniquely associated with each file in a system, and files are modeled as a class *File*, then *File* also denotes the set of all file names in a system.

File

$\ldots$

Let $\mathbb{O}$ denote the set of all object identities, regardless of class, that may be in the system. Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* in $\mathbb{O}$, it is with the understanding that we are in fact referring to the object whose identity is in $\mathbb{O}$.
Subsets of $\mathbb{O}$ can be formed by the union of sets of object identities, e.g. if A and B are distinct classes, $A \cup B$ is the collection of identities in $\mathbb{O}$ of objects of classes A or B. In this case, if we define $C \mathrel{\widehat=} A \cup B$ then C is not itself a class but only a set of object identities. C is called a *class union*. For a detailed discussion of class-union see [2].

2.2. Secondary Attributes

Attributes of a class are partitioned into primary and secondary. The values of the primary attributes determine the state of an object; the values of the secondary attributes depend upon the primary attributes of this or other objects in the system and enable an object to retain information about various aspects of the state of the system.
For example, consider classes *Bool* and *And*:

Bool

$val : \mathbb{B}$

various operations

And

$left, right : Bool \cup And$
Δ
$val : \mathbb{B}$

$val = left.val \land right.val$

various operations

The class *Bool* has one (primary) attribute, *val*. For the class *And*, objects *left* and *right* are primary attributes, while *val* is a secondary attribute whose value depends upon *left.val* and *right.val*. In Object-Z, the declaration of primary and secondary attributes are syntactically separated by the Δ symbol.
Secondary attributes are implicitly included in the Δ-list of every operation in the system. (The understanding of the Δ-list of an operation in Object-Z is that attributes which are so listed are subject to change. Secondary attributes are therefore always subject to change whenever an operation is invoked; primary attributes not included in the Δ-list remain unchanged.) In the above case, if a is an object of class *And*, the value of $a.val$ can change if either of the objects $a.left$ or $a.right$ undergoes an operation that changes the value of its *val* attribute, even if a itself undergoes no operation. For a detailed discussion of secondary attributes see [5].

2.3. Local Objects

When considering a system from the point of view of a particular object, it is often the case that various objects refer-

[1] Note that, the meaning of the program queries are more related to the design of SPAs, while the semantics of program query language belongs to a different level/issue which is more related to the semantics of SQL

enced by the given object can be considered as local to the object. These local object structures form an acyclic pattern. For example, inside an expression there may be other expressions, but an expression cannot be inside itself.
To capture these ideas within Object-Z, associated with each class is an implicit secondary attribute *dlocal* denoting the subset of $\mathbb{O}$ consisting of all local objects directly referenced by objects of the class. For example, suppose *A*, *B* and *C* are classes and *D* is defined by

D

$$a : A;\ b : B;\ sb : \mathbb{P}\, B;\ sc : \operatorname{seq} C$$

etc.

If the local objects in *D* are the object *a*, the objects in the set *sb* and the objects in the sequence *sc*, then

$$dlocal = \{a\} \cup sb \cup \operatorname{ran} sc.$$

Hence, if *ob* is an object of class *D*, *ob.dlocal* denotes the set of local objects directly referenced by *ob*, i.e.

$$ob.dlocal = \{ob.a\} \cup ob.sb \cup \operatorname{ran} ob.sc.$$

If the role of an attribute is to always identify a local object, this can be indicated when the attribute is declared by appending a subscript `$_{\copyright}$' to the appropriate type. This removes the necessity to write an explicit predicate to define the value of the implicit secondary attribute *dlocal*. For example, adopting this syntactic convention in the specification of the class *D* gives

D

$$a : A_{\copyright};\ b : B;\ sb : \mathbb{P}\, B_{\copyright};\ sc : \operatorname{seq} C_{\copyright}$$

etc.

Also associated with each class is an implicit secondary attribute *local* denoting the set of local objects directly or indirectly referenced by objects of the class. To be specific, for any object *ob*,

$$\begin{aligned} ob.local = \{obj : \mathbb{O} \mid \exists\, s : \operatorname{seq}_1 \mathbb{O} \bullet \\ s(1) \in ob.dlocal \wedge s(\#s) = obj \\ \forall\, i : 1 \mathinner{..} \#s - 1 \bullet s(i+1) \in s(i).dlocal\} \end{aligned}$$

i.e. $obj \in ob.local$ if there is a sequence of objects that starts with *ob* and ends with *obj*, and where each object in the sequence is a local object directly referenced by its predecessor object in the sequence.
The acyclic structure of the local object references can now be easily stated:

$$\forall\, ob : \mathbb{O} \bullet ob \notin ob.local$$

This predicate is an implicit global invariant of any system. When specifying the semantics of the COBOL language later in the paper, for example, this invariant will ensure that COBOL expressions do not contain themselves.
Local objects is one kind of object containment (sharable containment). For a detailed discussion of object containment (including local objects) see [3].

3. Formal Object Model of COBOL Static Semantics

The key idea of using an object-oriented semantic approach is to use a class to model each language construct of COBOL. That is, each language construct, such as expression, statement and program, is modeled by Object-Z classes, whereas traditionally it has been modeled as more primitive mathematical entities such as semantic functions or composite types (in VDM terms). Below, we present a formal static semantics for a subset of COBOL. The reason we chose a subset in this paper, not the whole COBOL language is because we would like to concentrate on the essential ideas of our approach rather than presenting a tedious semantics documentation for COBOL. However, this COBOL subset covers most of the essential features of the COBOL language.

3.1. Constants and Variables

To begin, as the type of a COBOL constant or variable or expression can be numeric, alphanumeric, alphabetic, record or boolean, *Type* is defined as a free type:

$$Type ::= Nm \mid AlphNm \mid Alph \mid Bool \mid Record$$

All constants and variables can be modeled as objects of the following classes.
A constant is either a number constant or a string constant which is modeled as a class union:

$$Const \mathrel{\widehat{=}} NmConstant \cup StrConstant$$

NmConstant

$$type : Type$$

$$type = Nm$$

StrConstant

$$type : Type$$

$$type \in \{AlphNm, Alph\}$$

Variables not only have a type but also have a name. A variable is either a elementary variable or a group (record) variable:

$$Var \mathrel{\widehat{=}} ElemVariable \cup GroupVariable$$

$$\begin{array}{|l}\hline \textit{ElemVariable} \\ \hline id : Id;\ type : Type \\ \hline type \in \\ \quad \{Nm, AlphNm, Alph\} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{GroupVariable} \\ \hline id : Id;\ type : Type \\ field : \mathrm{seq}\, Var \\ \hline type = Record \\ \hline \end{array}$$

3.2. Files

COBOL files are also modeled as objects of the classes:

$$\begin{array}{|l}\hline \textit{PhysicalFile} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{LogicalFile} \\ \hline record : Var \\ key : Id \\ \hline record.type = Record \\ \hline \end{array}$$

3.3. Expressions

A COBOL expression is either a constant, variable and binary expression. It can be modeled as a class-union:

$$Exp \mathrel{\widehat{=}} Const \cup Var \cup BinaryExp$$

where *Const* and *Var* are as defined in the previous section. The *BinaryExp* are specified in the following.
Each expression can be modeled as an object. For example, a binary expression is modeled as an object with two sub-expression(s).
The common binary expression frame can be defined as:

$$\begin{array}{|l}\hline \textit{BinaryFrame} \\ \hline l, r : Exp_{©};\ type : Type \\ \hline \end{array}$$

Classes *Equality*, *Conjunction*, *PlusExp* and *LessExp* of specific binary expressions are defined by inheriting the generic class *BinaryFrame*.

$$\begin{array}{|l}\hline \textit{Equality} \\ BinaryFrame \\ \hline type = Bool \\ l.type = r.type \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Conjunction} \\ BinaryFrame \\ \hline type = Bool \\ l.type = r.type = Bool \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{PlusExp} \\ BinaryFrame \\ \hline type = Nm \\ l.type = r.type = Nm \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{LessExp} \\ BinaryFrame \\ \hline type = Bool \\ l.type = r.type = Nm \\ \hline \end{array}$$

Note that all well typed information is captured by the state invariants of each class. The binary expressions can be defined as

$$BinaryExp \mathrel{\widehat{=}} Equality \cup Conjunction \cup PlusExp \cup LessExp \cup \ldots$$

Other binary expressions, such as `times', `disjunction', etc and unary expressions, can be similarly defined.

3.4. Statements

A COBOL statement is either an Assign-statement, a Control-statement or a File-statement:

$$Stmt \mathrel{\widehat{=}} AssignStmt \cup ControlStmt \cup FileStmt$$

An assign-statement is either a Add-to, Compute or Move-to statement:

$$AssignStmt \mathrel{\widehat{=}} AddTo \cup Compute \cup MoveTo$$

where *AddTo*, *Compute* and *MoveTo* are modeled as:

$$\begin{array}{|l}\hline \textit{AddTo} \\ \hline e : Exp_{©} \\ vs : \mathbb{P}_1\, Var_{©} \\ \hline \forall\, v : vs \bullet \\ \quad e.type = v.type \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Compute} \\ AddTo \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{MoveTo} \\ AddTo \\ \hline e \in Var \cup Const \\ \hline \end{array}$$

The statements *AddTo*, *Compute* and *MoveTo* have very similar static semantics, but different dynamic semantics. Note that the dynamic semantics of a programming language can be modeled in a class, see [4], however, it is unnecessary in this paper as the focus of this paper is SPAs rather than dynamic program analysers.
A Control-statement is either a Perform, Call or If statement:

$$ControlStmt \mathrel{\widehat{=}} Perform \cup Call \cup IfStmt$$

where *Perform*, *Call* and *IfStmt* are modeled as:

$$\begin{array}{|l}\hline \textit{Perform} \\ \hline proc : Proc \\ until : Exp \\ \hline until.type = Bool \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Call} \\ \hline prog : Prog \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{IfStmt} \\ \hline test : Exp_{©} \\ then, else : \mathrm{seq}\, Stmt_{©} \\ \hline test.type = Bool \\ \hline \end{array}$$

For simplicity, we model the Call-statement without parameters.
A File-statement is either a Open, Read or Write statement:

$$FileStmt \mathrel{\widehat{=}} Open \cup Read \cup Write$$

$$\begin{array}{|l} \text{Open} \\ \hline \mathit{file} : \mathit{LogicalFile} \end{array} \quad \begin{array}{|l} \text{Read} \\ \hline \mathit{file} : \mathit{LogicalFile} \end{array} \quad \begin{array}{|l} \text{Write} \\ \hline \mathit{var} : \mathit{Var} \\ \hline \exists \mathit{lf} : \mathit{LogicalFile} \bullet \\ \quad \mathit{lf}.\mathit{record} = \mathit{var} \end{array}$$

3.5. Procedure and Program

We take a simple view of representing COBOL's `sections' and `paragraphs' as procedures.
Just as with expressions and statements, a procedure or a program is modeled as an object. As procedures are similar in structure to programs, the common features are specified by an auxiliary class *Block* which is inherited by the classes *Proc* and *Prog*.
A *Block* has a name, a body which is a sequence of statements and may also have a parent program.

$$\begin{array}{|l} \mathit{Block} \\ \hline \mathit{name} : \mathit{Id} \\ \mathit{body} : \mathrm{seq}\, \mathit{Stmt}_{©} \\ \mathit{parent} : \mathit{Prog} \cup \mathit{Nil} \\ \Delta \\ \mathit{varscope} : \mathbb{P}\, \mathit{Var} \\ \mathit{procscope} : \mathbb{P}\, \mathit{Proc} \\ \mathit{progscope} : \mathbb{P}\, \mathit{Prog} \\ \hline \mathit{Var} \cap \mathit{local} \subseteq \mathit{varscope} \\ \{p : (\mathit{Perform} \cap \mathit{local}) \bullet p.\mathit{proc}\} \subseteq \mathit{procscope} \\ \{p : (\mathit{Call} \cap \mathit{local}) \bullet p.\mathit{prog}\} \subseteq \mathit{progscope} \end{array}$$

where

$$\begin{array}{|l} \mathit{Nil} \\ \hline \end{array}$$

representing a nil class which has no content.

$$\begin{array}{|l} \mathit{Proc} \\ \mathit{Block} \\ \hline \mathit{parent} \in \mathit{Prog} \\ \mathit{varscope} = \mathit{parent}.\mathit{varscope} \\ \mathit{procscope} = \mathit{parent}.\mathit{procscope} \\ \mathit{progscope} = \mathit{parent}.\mathit{progscope} \end{array}$$

The secondary attribute *varscope* denotes the set of all variables that are in scope. Again these are modeled as variable objects. For a procedure, this set is formed by the set of variables (in scope) of the parent program. The secondary attribute *procscope* denotes the set of all procedures that are in scope. Again these are modeled as procedure objects. For a procedure, this set is formed by the set of (locally defined) procedures of the parent program.
A COBOL program has additional information, such as the relation between logical files and physical files *assign*, the locally declared variables *vars*, a set of procedures *procs* and a set of sub-programs *subprogs*.

$$\begin{array}{|l} \mathit{Prog} \\ \mathit{Block} \\ \hline \mathit{assign} : \mathit{LogicalFile}_{©} \rightarrow \mathit{PhysicalFile}_{©} \\ \mathit{vars} : \mathbb{P}\, \mathit{Var} \\ \mathit{procs} : \mathbb{P}\, \mathit{Proc} \\ \mathit{subprogs} : \mathbb{P}\, \mathit{Prog} \\ \hline \forall p : (\mathit{procs} \cup \mathit{subprogs}) \bullet p.\mathit{parent} = \mathit{self} \\ \mathit{procscope} = \mathit{procs} \\ \mathit{varscope} = \mathit{vars} \\ \mathit{parent} \in \mathit{Nil} \Rightarrow \mathit{progscope} = \\ \qquad \{p : \mathit{Prog} \mid p.\mathit{parent} \in \mathit{Nil}\} \oplus_p \mathit{subprogs} \\ \mathit{parent} \in \mathit{Prog} \Rightarrow \mathit{progscope} = \\ \qquad \mathit{parent}.\mathit{progscope} \oplus_p \mathit{subprogs} \end{array}$$

For a program, the *varscope* is formed by the (locally declared) variables of that program. The *procscope* is formed by the (locally defined) procedures of that program.
For a top-program, the *progscope* is formed by the locally declared sub-programs override all other top-programs (if there is a name clash), where the override operator `$\oplus_p$' is defined by

$$\begin{array}{|l} \_ \oplus_p \_ : \mathbb{P}\, \mathit{Prog} \times \mathbb{P}\, \mathit{Prog} \rightarrow \mathbb{P}\, \mathit{Prog} \\ \hline \forall s_1, s_2 : \mathbb{P}\, \mathit{Prog} \bullet s_1 \oplus_p s_2 = \\ \qquad (s_1 \setminus \{p_1 : s_1 \mid \exists p_2 : s_2 \bullet p_1.\mathit{id} = p_2.\mathit{id}\}) \cup s_2 \end{array}$$

For a sub-program the *progscope* is obtained by having the locally declared programs override the set of programs inherited from the parent block

4. Formally Capturing Code Relationships

From the static semantics of the COBOL language, we can derive many useful formal relationships between language constructs. For example,
The hierarchy relationship between programs and subprograms can be captured as:

$$\begin{array}{|l} \mathit{desc\_progs} : \mathit{Prog} \rightarrow (\mathbb{P}\, \mathit{Prog}) \\ \hline \forall p_1 : \mathit{Prog} \bullet \mathit{desc\_progs}(p_1) = \\ \qquad \bigcup \{p_2 : p_1.\mathit{subprogs} \bullet \mathit{desc\_progs}(p_2)\} \cup p_1.\mathit{subprogs} \end{array}$$

The hierarchy relationship between programs and procedures can be captured as:

$$
\begin{array}{|l}
\mathit{desc\_procs} : \mathit{Prog} \rightarrow (\mathbb{P}\, \mathit{Proc}) \\ \hline
\forall \mathit{pg} : \mathit{Prog} \bullet \\
\qquad \mathit{desc\_procs}(\mathit{pg}) = \bigcup\{p : \mathit{desc\_progs}(\mathit{pg}) \bullet p.\mathit{procs}\}
\end{array}
$$

The hierarchy relationship between programs and variables can be captured as:

$$
\begin{array}{|l}
\mathit{desc\_vars} : \mathit{Prog} \rightarrow (\mathbb{P}\, \mathit{Var}) \\ \hline
\forall \mathit{pg} : \mathit{Prog} \bullet \\
\qquad \mathit{desc\_vars}(\mathit{pg}) = \bigcup\{p : \mathit{desc\_progs}(\mathit{pg}) \bullet p.\mathit{vars}\}
\end{array}
$$

The direct call graphs between programs can be captured as:

$$
\begin{array}{|l}
\mathit{calls} : \mathit{Prog} \leftrightarrow \mathit{Prog} \\ \hline
\forall p_1, p_2 : \mathit{Prog} \bullet \\
\qquad p_1 \ \underline{\mathit{calls}}\ p_2 \Leftrightarrow p_2 \in \{p : (p_1.\mathit{local} \cap \mathit{Call}) \bullet p.\mathit{prog}\}
\end{array}
$$

The direct and indirect call graphs can be modeled by calls^+ which is a transitive closure[2] of *calls*.
The relationship between procedures and their corresponding modified variables can be captured as:

$$
\begin{array}{|l}
\mathit{modifiedBy} : \mathit{Proc} \rightarrow (\mathbb{P}\, \mathit{Var}) \\ \hline
\forall p : \mathit{Proc} \bullet \mathit{modifiedBy}(p) = \{v : \mathit{Var} \mid \\
\qquad \exists\, \mathit{stmt} : (\mathit{AssignStmt} \cap p.\mathit{local}) \bullet v \in \mathit{stmt}.\mathit{vs}\}
\end{array}
$$

Even matching expression patterns can be captured. For example, two expression patterns that share the same concrete syntax can be modeled as:

$$
\begin{array}{|l}
\mathit{same} : \mathit{Exp} \leftrightarrow \mathit{Exp} \\ \hline
\forall e_1, e_2 : \mathit{Exp} \bullet e_1 \ \underline{\mathit{same}}\ e_2 \Leftrightarrow \\
\ \{e_1, e_2\} \subseteq \mathit{Const} \wedge e_1 = e_2 \ \vee \\
\ \{e_1, e_2\} \subseteq \mathit{Var} \wedge e_1.\mathit{type} = e_2.\mathit{type} \wedge e_1.\mathit{id} = e_2.\mathit{id} \ \vee \\
\ (\{e_1, e_2\} \subseteq \mathit{Equality} \vee \\
\ \ \{e_1, e_2\} \subseteq \mathit{PlusExp} \vee \\
\ \ \{e_1, e_2\} \subseteq \mathit{Conjunction} \vee \\
\ \ \{e_1, e_2\} \subseteq \mathit{LessExp}) \wedge \\
\ \ e_1.l \ \underline{\mathit{same}}\ e_2.l \wedge e_1.r \ \underline{\mathit{same}}\ e_2.r
\end{array}
$$

[2] In general, if $R : X \leftrightarrow X$, then forward relational composition

$$R \mathbin{⨾} R == \{x, z : X \mid \exists\, y : X \bullet x \ \underline{R}\ y \wedge y \ \underline{R}\ z\}$$

The homogeneous relation R composed with itself k times: $R^0 = idX$ and $R^{k+1} = R^k \mathbin{⨾} R$. Finally, the transitive closure $R^+ == \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$.

In this section, we demonstrated that based on the formal semantics of COBOL (presented in Section 3), we can precisely and elegantly capture useful generic information about code. These generic formal structures not only facilitate the formalisation of program queries and the design of an SPA in the next section, but also provide some formal foundations for designing program query languages. For example, in [8](Section 4), instances of PQL to answer the query

- “Find variables that are modified by a procedure programs that are called from more than 10 other programs”
- “Find variables that are modified by a procedure”

can be rigorously formulated based on the formal relation *calls* and *modifiedBy* respectively:

- **Select** p (program) **such that** $\#(\mathit{calls} \rhd \{p\}) > 10$
- **Select** v (variable) **such that** $v \in \mathit{modifiedBy}(\mathit{proc})$

5. Formalising Program Queries: A Formal Design of SPA

We can model an SPA as an object with operations which precisely capture the relationship between the input and expected output of generic program queries. For example, the following is a list of generic program queries about global program design issues:

- Given a physical file, which programs refer to this physical file?
- Given a access key, which physical files are accessed by the key?
- Given a procedure, which variables are modified by the procedure?
- Given a program, which programs are called from it?

Based on formalisations from Section 4, we can now specify the above queries and provide a design model of an SPA as follows:

SPA_GlobalProgram

$top\_progs : \mathbb{P}_1\ Prog_{©}$ [given programs]

Δ

$progs : \mathbb{P}_1\ Prog$ [all programs]

$procs : \mathbb{P}_1\ Proc$ [all procedures]

$vars : \mathbb{P}_1\ Var$ [all variables]

$$progs = \bigcup\{p : top\_progs \bullet desc\_progs(p)\}$$
$$procs = \bigcup\{p : top\_progs \bullet desc\_procs(p)\}$$
$$vars = \bigcup\{p : top\_progs \bullet desc\_vars(p)\}$$
$$\forall\, call : (Call \cap local) \bullet call.prog \in progs$$
$$\forall\, perf : (Perform \cap local) \bullet perf.proc \in procs$$
$$\forall\, var : (Var \cap local) \bullet var \in vars$$

GivenFile_FindPrograms

$file? : PhysicalFile;\ \ progs! : \mathbb{P}\ progs$

$$progs! = \{p : progs \mid file? \in p.local\}$$

GivenKey_FindFiles

$key? : Id;\ \ files! : \mathbb{P}\ PhysicalFile$

$$\begin{aligned} files! = \{&pf : PhysicalFile \mid \\ &\exists\, p : progs;\ lf : LogicalFile \bullet \\ &\quad p.assign(lf) = pf \wedge lf.key = key?\} \end{aligned}$$

GivenProc_FindModifiedVars

$proc? : procs;\ \ vars! : \mathbb{P}\ vars$

$$vars! = modifiedBy(proc?)$$

GivenProg_FindCalledProgs

$caller? : progs;\ \ called! : \mathbb{P}\ progs$

$$called! = calls(\!|\ \{caller?\}\ |\!)$$

Reload_SourceCode

$\Delta(top\_progs);\ \ newinput? : \mathbb{P}\ Prog$

$$top\_progs' = newinput?$$

For the operation *GivenProg_FindCalledProgs*, the relationship between input *caller* and output *called*! of the query is elegantly captured by the Z mathematical toolkit routine, the relation image [3].

Note that this design model also formally states what a complete and valid input source code space is by using class invariants. Therefore when a new source code space is loaded (*Reload_SourceCode*), a precondition is to satisfy all the invariant properties.

5.1. Extendibility

Whenever we need a new type of query, we can simply use inheritance to extend the model of the SPA. Suppose we want to answer some queries relating to program structure or detailed program design issues, for example, to be able to answer the following queries:

- Given a sub-expression pattern, which Compute statements contain this pattern?
- Given two variables, which statements modify the first and use the second at the same time?

SPA_Add_ProgramStructure

SPA_GlobalProgram

GivenExp_FindComputeStmts

$e? : Expression$

$comps! : \mathbb{P}\ Compute$

$$comps! = \{s : Compute \mid \exists\, e : s.e.local \bullet e\ \underline{same}\ e?\}$$

Given2Vars_FindStmtsM1U2

$v_1?, v_2? : Var$

$stmts! : \mathbb{P}\ Stmt$

$$\begin{aligned} stmts! = \{stm : AssignStmt \mid &\exists\, lv_1, lv_2 : stm.local \bullet \\ &lv_1\ \underline{same}\ v_1? \wedge lv_2\ \underline{same}\ v_2? \\ &lv_1? \in stm.vs \wedge lv_2? \in stm.e.local\} \end{aligned}$$

In this section, we presented a formal abstract model of an SPA whose operations model program queries. In addition, we demonstrated that the model can be extended (by inheritance) to include any new type of queries.

5.2. Discussions

The above query specifications form a reference design model for systematic development of SPAs. We found such a reference model useful in the following ways. Firstly, the model facilitates communication between query language designers and SPA designers, clarifying any doubts about the intended meaning of the query language constructs. We found it is more useful to specify a representative set of generic queries rather than to investigate the semantics of PQL constructs. Because the semantics of PQL constructs are more related to the semantics of general query languages (like SQL) which were well studied.

[3] Given $R : X \leftrightarrow Y$, the image of the set of S through the relation R is defined as:

$$R(\!|\ S\ |\!) == \{y : Y \mid \exists\, x : S \bullet x\ \underline{R}\ y\}$$

Secondly, formal semantics of queries provides guidelines for implementing a query evaluator [8], which is the most complex component of an SPA. A query evaluator accepts a query in a parsed, internal form and based on the program information, computes the result, i.e., selects program elements that match conditions described by the query. A query evaluator transforms a query to enable efficient query evaluation. The reference model given by formally specified queries allows the designers to validate each transformation to ensure that the optimisation process does not change the meaning of a query.

Generally, efficient evaluation of queries for program query languages [1, 8, 10] is still an open research problem and the main obstacle in wide-spread application of static program analyzers for large software systems. A query evaluator applies optimising transformations on case-by-case basis, for specific classes of queries such as those listed at the beginning of this section. Formal specification of queries abstract away syntactic differences between queries and emphasis the meaning. Syntactically different queries may be still semantically equivalent which may lead to discovering more optimising transformations and implementing more efficient query evaluators.

6. Conclusion

It is difficult to rigorously design an SPA based on informal defined program queries. In this paper, we present a formal object-oriented semantic approach to designing SPAs for programming languages. The essential steps of this approach are to:

(1) define an object-oriented model of the static semantics of the programming language constructs;

(2) derive formal relationships between program objects based on (1);

(3) formally design an SPA based on (1) and (2).

Using this approach, this paper presents a formal object model of the static semantics of a COBOL subset. The semantic model is reused to define a number of formal relationships between COBOL program objects. Based on this formal semantics model and the formal relationships, a COBOL SPA is modeled by an Object-Z class whose operations formally define various generic program queries. Object-Z notation was chosen as the meta language mainly because its semantics is well documented. Other object-oriented extensions to Z/VDM or even Concept Mapping Language (CML) [9] perhaps can also be applied in this approach.

The extendibility of the formal model of SPAs is also demonstrated. This work provides a formal base for designing program query languages. The formal design of SPAs concentrated on modelling the precise relationship between the input and the expected output of each program query, without concern as to how to compute the query. This separation of concerns allows our formal model to be more generic and more abstract, which can be implemented in various ways.

This paper demonstrates that program queries can be formally specified and suggest the ways in which specifications can be useful in SPA design and how specifications affect the quality of an SPA. Formal specification languages (in this case Object-Z) can be an elegant way of defining SPA functionality prior to its implementation. Further work will be to investigate the semantics-directed SPA generation which is to derive the implementation based on our formal design. We also plan to extend this approach to formally design dynamic program analysers by firstly adding dynamic semantics to each class model, then using operations to capture the dynamic program queries.

References

[1] S. Burson, G. Kotik, and L. Markosian. A program transformation approach to automating software re-engineering. In *COMPSAC'90*, pages 314–322, 1990.

[2] J.S. Dong and R. Duke. Class Union and Polymorphism. In *the 12th International Conference on Technology of Object-Oriented Languages and Systems*, pages 181–190. Prentice-Hall, 1993.

[3] J.S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems (OOS)*, 2(1):41–63, Chapman & Hall 1995.

[4] J.S. Dong, R. Duke, and G. Rose. An Object-Oriented Denotational Semantics of A Small Programming Language. *OOS*, 4(1):29–52, 1997.

[5] J.S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. In *the IEEE International Conference on Engineering Complex Computer Systems*, pages 31–38, 1995. IEEE Press.

[6] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[7] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *OOS*, 2:195–215, 1995.

[8] S. Jarzabek. Design of Flexible Static Program Analyzers with PQL. *IEEE Trans. Software Eng.*, pages 197–215, March 1998.

[9] W.M. Lim, J. Harrison, P. Bailes, and A. Berglas. Design Recovery Through Formal Specification. In *Australian Software Engineering Conference*, Nov 1998. IEEE Press.

[10] S. Paul and A. Prakash. A query algebra for program databases. *IEEE Trans. Software Eng.*, 22(3):202–217, March 1996.