A Knowledge-based Program Transformation System

P. C-Y. Sheu
Department of Electrical & Computer Engineering
University of California
Irvine, CA 92717

S. Yoo
Department of Industrial Automation
Inha University
Souel, Korea

Abstract. This paper describes a Knowledge-Based Program Transformation System (KBPTS) that has been designed on top of an object-oriented knowledge base for the purpose of automatic program transformation and optimization. In KBPTS, a program can be specified by means of a flowchart or a set of logical descriptions. Generally, given a specification of a program, it can be synthesized with a set of procedural methods. However, a simple substitution of a method for a basic computation in a specification may leave a large amount of possible optimizations unexplored. Assuming that a set of efficient algorithms for abstract problems (e.g., graph algorithms) is implemented and saved in an object-oriented knowledge base, a given program (or parts of it) can be evaluated by those algorithms efficiently with proper instantiations of variables. To identify the proper algorithms, the conventional unification algorithm has been modified into an *analogical unification* algorithm. Also, in order to control the overall search space more clearly, a set of global search strategies are encoded in meta rules.

1. Introduction

This paper describes a Knowledge-Based Program Transformation System (KBPTS) that has been designed on top of an object-oriented knowledge base for the purpose of automatic program transformation and optimization. In KBPTS, a program can be specified by means of a flowchart or a set of logical descriptions. A flowchart is a set of nodes and directed edges, where each node represents some computational actions and each edge designates a possible passage of control between nodes. In an object-oriented knowledge base, the computational actions for each node can be considered as a predefined method. A set of logical descriptions specifies the precondition and the postcondition of a program which correspond to the functionality of the program. Generally, given a specification of a program, it can be synthesized with a set of procedural methods. However, a simple substitution of a method for a basic computation in a specification may leave a large amount of possible optimizations unexplored. Assuming that a set of efficient algorithms for abstract problems (e.g., graph algorithms) is implemented and saved in an object-oriented knowledge base, a given program (or parts of it) can be evaluated by those methods efficiently with proper instantiations of variables.

This transformation system works as an assertion-based logical system [Floy67] [Hoar69]. In an object-oriented knowledge base, a set of methods is associated to each object class, and the definition of each method includes a precondition and a postcondition. If a method's precondition is true before its execution, the postcondition will be true upon its termination. These assertions do not necessarily assign particular values to each variable, but they specify some properties of the values or any relationships among them. The object-level knowledge base also includes axioms and rules of inference specified in first-order predicate calculus, which can be applied to transform a program according to the global strategies defined as meta rules. The meta rules are not domain specific; they specify a set of general control criteria which guides the overall search process [DaBu84].

The paper is organized as follows. Section 2 overviews the architecture of KBPTS. Section 3 describes the structure and the internals of the algorithm library.

Section 4 discusses an mechanism that can be applied to substitute program specifications with library algorithms. Section 5 sketches some global control strategies to reduce the search space. Section 6 compares KBPTS with some related work. Finally, Section 7 concludes the paper with some future research directions.

2. Overviews of KBPTS

In [Zani84] [Sheu88], mathematical logic has been employed to describe *objects*, *classes*, and *methods*. In the object-oriented knowledge base framework, the first-order language consists of the following [Sheu88]:

(1) *class(a)* is true if a is the name of a class of objects.
(2) *instance_of(a,b)* is true if object a is a member of class b.
(3) *subclass_of(a,b)* is true if class a is a subclass of class b.
(4) *attribute(a,b,c)* is true if every object in class a has b as one of its attributes, and the domain of b is c.
(5) *attribute_value*(a,b,c) is true if object a has the value c of its attribute b.
(6) $a(x_1,...,x_r)$ is true if a is a method which has the arguments $x_1,...,x_r$. We call $a(x_1,...,x_r)$ a *method predicate*.
(7) $f(x_1,...,x_n)$ is true if a tuple $<x_1, ... , x_n>$ is a member of user-defined relation f. In this case, we call $f(x_1,...,x_n)$ a *relational predicate*.

With this first order language, database schemata, deductive laws and integrity constraints can be defined. The key components of KBPTS can now be described in the following:

(a) *Object Schema:*

The object schema defines classes in predicate forms, where the definition of a class includes the name, the attributes, the methods, and the class hierarchy. Suppose we have a set of object classes related to graphs, i.e., *vertex*, *edge* and *weighted_graph*. Assuming the length of each edge is available, the class *weighted_graph* can have the procedure *shortest_path* as a method. The method *shortest_path* searches a given graph and returns a shortest path between two given vertices. The definition of this database in the object schema can be summarized as follows:

- class(vertex)
  - *attribute*(*vertex*,*VID*,*string*)
  - *attribute*(*vertex*,*X_POS*,*real*)
  - *attribute*(*vertex*,*Y_POS*,*real*)
- class(edge)
  - *attribute*(*edge*,*EID*,*string*)
  - *attribute*(*edge*,*V1*,*vertex*)
  - *attribute*(*edge*,*V2*,*vertex*)
  - *attribute*(*edge*,*WEIGHT*,*integer*)
- class(weighted_graph)
  - *attribute*(*weighted_graph*,*GID*,*string*)
  - *attribute*(*weighted_graph*,*V*,*set−of−vertex*)
  - *attribute*(*weighted_graph*,*E*,*set−of−edge*)

□ method
shortest_path(V1:vertex, V2:vertex, G:weighted_graph, P:path)

(b) *Data Base:*

Objects in a class can be stored in the form of a table, where each tuple corresponds to an object instance in the class. A class consists of a number of attributes, and the value of an attribute is an object or a set of objects which belongs to some other class. This relationship among attributes and classes forms a *class–attribute* hierarchy [BeKi89].

(c) *Rule Base:*

In the rule base, we have both deductive laws and integrity constraints in the following form without function symbols:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow Q_1 \vee \cdots \vee Q_m$$

which can be interpreted as " if all of P_1, P_2, ..., and P_n are true, then at least one of Q_1, Q_2, ..., or Q_m is true ".

(d) *Meta Rule Base:*

While the rules in the rule base (i.e., deductive rules and integrity constraints) describe the knowledge about the objects in the database, meta rules represent the knowledge about the object-level knowledge. For example, the meta rule base can include a set of rules of cost estimation so that a single rule can be selected to apply among several possibilities. As a result, meta rules form a set of strategies to guide the use of object-level knowledge by pruning or reordering the rules in the rule base [DaBu84].

(e) *Knowledge Base Management System:*

The main functions of the knowledge base management system are query optimization, transaction processing, and object management. In addition to queries, insert and delete operations can be included in a transaction. To process a transaction, data dependency among queries and other operations needs to be analyzed for optimization. The management system also provides other amenities such as concurrency control, integrity validation, crash recovery, and authorization.

(f) *User Interface:*

The system interacts with the user in various methodologies, such as flowcharts, logical descriptions and database programs [ShYo90]. In this paper, we assume that a program is presented as an annotated flow chart as described in [Floy67] and [Hoar69]. In [Floy67], Floyd associated an assertion with each arc in a flowchart such that if the assertion P associated with the entrance arc of an action π is true before π is executed then the assertion R associated with the exit arc of π will be true after π is executed. Using the notation in [Hoar69], this can be represented as

$$\{P\}\ \pi\ \{R\},$$

where P and R are usually mentioned as *precondition* and *postcondition*, respectively, of process π.

3. A Library of Algorithms

In an object-oriented knowledge base, for each object class, a set of methods can be associated. We assume that a large number of abstract algorithms are coded as methods in an object-oriented knowledge base. These methods usually belong to abstract object classes such as *graph* and other data structures. Clearly, it will be

desirable if a programmer can reuse some of these algorithms deliberately or with some help from the system.

We consider a library of algorithms as a collection of useful methods. In order to be selected properly for substitutions, the definition of each method should include the following:

(a) method name
(b) a list of arguments, where the domain of each argument should be specified
(c) the functionality of the method in terms of its precondition and postcondition
(d) order of complexity

As an example, we can present a set of object classes related to the class *graph* and some methods which implement efficient algorithms. In the following example, we use P, Q and O to denote a precondition, a postcondition, and the order of complexity of a method, respectively.

object classes and methods

<class>
vertex(VID:integer)
edge(EID:integer, V1:vertex, V2:vertex)
vertex_pair(VPID:integer, V1:vertex, V2:vertex)
weighted_edge(WEID:integer, V1:vertex, V2:vertex, W:integer)
path(PID:integer, V1:vertex, V2:vertex, E:set-of-edge)
weighted_path(WPID:integer, V1:vertex, V2:vertex, E:set-of-weighted_edge)
graph(GID:integer, V:set-of-vertex, E:set-of-edge)

<subclass>
weighted_graph(E:set-of-weighted_edge)

; set all the possible paths between V1 and V2 to SP
<method> (all_path (V1:vertex) (V2:vertex) (V:set-of-vertex)
(E:set-of-weighted_edge) (SP:set-of-weighted_path) ...)
P = { *true* }
Q = { $(\forall P)$ [*member_of(P, SP)* $\rightarrow$ *weighted_path(WP1, V1, V2, P.E)*] }
O = *cardinality_of(V)* + *cardinality_of(E)*

; return *true if a given graph* (*V*,*E*) is connected
; the returned value is denoted by *RETURN*
; in the following logical descriptions
<method> (connected_graph (V:set-of-vertex) (E:set-of-weighted_edge) ...)
P = { *true* }
Q = { [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$ *member_of(Y, V)*
$\wedge$ *weighted_path(WP1, X, Y, E)*] $\rightarrow$ *RETURN*] $\vee$
[~ [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$
member_of(Y, V) $\wedge$ *weighted_path(WP1, X, Y, E)*]] $\rightarrow$
~ *RETURN*] }
O = *cardinality_of(V)* + *cardinality_of(E)*

;return *true if a given graph* (*V*,*E*) *is a tree*
<method> (is_tree (V:set-of-vertex) (E:set-of-weighted_edge) ...)
P = { *true* }
Q = { [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$
member_of(Y, V) $\wedge$ *all_path(X, Y, V, E, SP)* $\wedge$
equal(cardinality_of(SP), 1)] $\rightarrow$ *RETURN*] $\vee$
[~ [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$

member_of(Y, V) ∧ all_path(X, Y, V, E, SP) ∧
equal(cardinality_of(SP), 1)]] → ~ RETURN] }
O = cardinality_of(V) + cardinality_of(E)

;set the total weight of a weighted path to *SW*
<method> (sum_of_weight (WP:weighted_path, SW:integer) ...)
P = { true }
Q = { equal(SW, summation(W, WP.E)) }
O = cardinality_of(WP.E)

;set the spanning tree with the minimum weight to *MST*
<method> (minimum_spanning_tree (G:weighted_graph,
MST:weighted_graph) ...)
P = { true }
Q = { (∀ X) [spanning_tree(X,G) ∧ weight_of_graph(X, WX) ∧
weight_of_graph(MST, WS) →
lesseq(WS, WX)] }
O = cardinality_of(G.V)²

;set the path between given two vertices with the minimum weight to *SP*
<method> (shortest_path (G:weighted_graph, V1:vertex,
V2:vertex, SP:weighted_path) ...)
P = { true }
Q = { (∀ X) [weighted_graph(G, V, E) ∧ all_path(V1, V2, V, E, P_SET)
∧ member_of(X, P_SET) ∧ sum_of_weight(SP, WSP) ∧
sum_of_weight(X, WX) → lesseq(WSP, WX)] }
O = cardinality_of(G.V)²

<subclass>
directed_graph(E:set-of-vertex_pair)

;set all the vertices which can be reached from a given vertex to *TC*
<method> (transitive_closure (G:directed_graph, V1:vertex,
TC:set-of-vertex) ...)
P = { true }
Q = { (∀ X) [member_of(X, TC) ∧ directed_graph(G, V, E) ∧
all_directed_path(V1, X, V, E, P_SET) →
greater(cardinality_of(P_SET), 0)] }
O = cardinality_of(G.V)²

;remove any directed cycle in a given directed graph
<method> (topological_sorting (G:directed_graph,
TS:list-of-vertex) ...)
P = { true }
Q = { (∀ X) (∀ Y) [member_of(X, TS) ∧ member_of(Y,TS) ∧
vertex_pair(VP, X, Y) → precede(X, Y, TS)] }
O = cardinality_of(G.V) + cardinality_of(G.E)

We can classify methods according to the way in which they return their results. Some of the methods return *true* or *false* according to their evaluation; we call these *logical* methods (e.g., *is_tree* and *connected_graph*). Other methods which are not logical methods are considered as *general* methods. In the above example, the returned values from logical methods are denoted by a special variable *RETURN* in the description of postconditions. The treatment of *RETURN* during the process of program transformation will be discussed later in this paper. The meanings of some predicates which have not been defined in the above example are briefly explained in the following:

(a) *member_of(A, S)*: which is true if object A is a member of a set S.
(b) *cardinality_of(S)* : which returns the cardinality of a set S.
(c) *equal(A, B)* : which is true if A is equal to B.
(d) *summation(A, B)*: which returns the sum of A for all the member of a set B.
(e) *lesseq(A, B)*: which is true if A is less than or equal to B.
(f) *greater(A, B)*: which is true if A is greater than B.
(g) *precede(A, B, L)*: which is true if A precede B in a list L.
(h) *all_directed_path(V1,V2,V,E,DP)*: which set all the possible directed paths from $V1$ to $V2$ to DP.
(i) *set-of-A*: denotes an object domain which is a set of objects of class A.
(i) *list-of-A*: denotes an object domain which is a list of objects of class A.
(j) *A.B*: denotes an attribute B of an object A.

4. Substituting Library Algorithms For Program Specification

Substituting a library algorithm for a part of program specification can be done by comparing the assertion of the specification part with that of the pre-stored methods. The procedure of this comparison is similar to that of a unification process in predicate calculus [Nils80]. A unification process is to find a set of substitutions which can be applied to two or more expressions and make them the same substitution instances. For example, consider two expressions: $E_1 = p(X,Y,a)$ and $E_2 = p(b,Y,Z)$. A substitution s can be applied to an expression E; the result of a substitution is an *substitution instance* and it is denoted by Es. If there exists an substitution s_i such that $E_1 s_i = E_2 s_i$, then E_1 and E_2 are *unifiable* and s_i is an *unifier*. In the above example, we can identify the following substitution instances:

$E_1 s = p(X,Y,a) \{ b/X, a/Z \} = p(b, Y, a)$
$E_2 s = p(b,Y,Z) \{ b/X, a/Z \} = p(b, Y, a)$

Because $E_1 s = E_2 s$, $p(X,Y,a)$ and $p(b,Y,Z)$ are unifiable with a unifier $\{ b/X, a/Z \}$.

A unification process can be considered as a pattern matching problem with substitutions of arguments. Several unification algorithms have been proposed, and it was proved that two expressions can be determined to be unifiable or not [Robi65]. However, most of the existing algorithms only allow a matching between two predicates with the same head. For this reason, using the conventional algorithms only allows the program transformation process be performed within an object class. For example, suppose that a detailed program has been designed in order to find a shortest path between vertices a and b. The program is written in a flowchart language, and suppose that the assertion for an edge which is incident on the *Stop* node has been derived as follows:

$$A = \{ (\forall X) [weighted\_graph(G, V, E) \wedge all\_path(a, b, V, E, P\_SET) \wedge member\_of(X, P\_SET) \wedge sum\_of\_weight(SP, WSP) \wedge sum\_of\_weight(X, WX) \rightarrow lesseq(WSP, WX)] \}$$

Clearly, Assertion A and the postcondition of the method *shortest_path* in class *graph* are unifiable if we substitute a for $V1$ and b for $V2$. In other words, assertion A is a special case for the postcondition of the method *shortest_path*, where *shortest_path* is a general function which finds a shortest path between any two vertices in a graph. Intuitively, the above unification is an example of instantiating variables (e.g., $V1$ and $V2$) with constants (e.g., a and b).

Now, consider a program written for an airline reservation system. The program finds the cheapest connection between two cities c_1 and c_2. Suppose that, after

applying the transformation rules to the program, we have the following assertion:

B = { $(\forall Y)$ [*airline(F, CITIES, FLIGHTS)* $\wedge$ *all_connection*(c_1, c_2, *CITIES, FLIGHTS, C_SET)* $\wedge$ *member_of(Y, C_SET)* $\wedge$ *sum_of_fare(SC, FSC)* $\wedge$ *sum_of_fare(Y, FY)* $\rightarrow$ *lesseq(FSC, FY)*] }

If we compare this assertion with the postcondition of the method *shortest_path* in class *graph*, we can find the following terms which syntactically correspond to each other:

weighted_graph(G, V, E)	*airline(F, CITIES, FLIGHTS)*
all_path(V1, V2, V, E, P_SET)	*all_connection*(c_1, c_2, *CITIES, FLIGH*
member_of(X, P_SET)	*member_of(Y, C_SET)*
sum_of_weight(SP, WSP)	*sum_of_fare(SC, FSC)*
sum_of_weight(X, WX)	*sum_of_fare(Y, FY)*
lesseq(WSP, WX)	*lesseq(FSC, FY)*

In conventional unification algorithms, two predicates which have different predicate heads (e.g., *weighted_graph(G, V, E)* and *airline(F, CITIES, FLIGHTS)*) cannot be unified. However, we know that the *shortest_path* algorithm in class *weighted_graph* can be used for finding the cheapest connection in the class *airline* by properly instantiating the variables in the *shortest_path* algorithm with those in the airline reservation system. This is an example of matching with analogy, and in order to perform this we need an *analogical* unification process. The term "analogical" has been used in various contexts in the artificial intelligence community (e.g., "analogical problem solving" [Carb81] and "programming by analogy" [Ders86]). While an analogical approach can include some interactive modification processes in order to use an existing program for a new problem, in this paper, the term "analogy" is strictly limited to exact correspondences between object classes or between programs. In this way, we can automate the process of program transformations.

In the above list of correspondences, each predicate is either a relational predicate or a method predicate. Here, a relational predicate represents the membership in a class, i.e., *weighted_graph(G, V, E)* is true if an instance of G, V and E is a member of the class *weighted_graph*. In order to compare two classes, we first identify the pairs of corresponding attributes, which should be kept consistent throughout the entire process. To be matched analogically, the domains of attributes in a pair should be the same. If an attribute is a compound object (i.e., it is a collection of other objects), the object structures (i.e., attributes and their domains) should be the same recursively. For example, the attribute *FLIGHTS* is a compound attribute and it specifies each *FLIGHT* with *FARE*, where *FLIGHT* corresponds to *EDGE* and *FARE* matches *WEIGHT*. As for method predicates, their unifiability should be determined by comparing the assertions of their functionality (i.e., their preconditions and postconditions) recursively.

During this unification process, the special variable *RETURN* which denotes the returned value from a *logical* method can match either a variable or a predicate. As an example, we consider a logical method *is_tree*. Suppose that we have a program which contains a variable *Tree* that is set to zero initially. After the main procedure is executed, *Tree* has the value one if a given graph is a tree. In this case, the assertion C can be given as follows:

C = { [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$ *member_of(Y, V)* $\wedge$ *all_path(X, Y, V, E, SP)* $\wedge$ *equal(cardinality_of(SP), 1)*] $\rightarrow$ *equal(Tree,1)*] $\vee$ [$\sim$ [$(\forall X)$ $(\forall Y)$ [*member_of(X, V)* $\wedge$ *member_of(Y, V)* $\wedge$ *all_path(X, Y, V, E, SP)* $\wedge$ *equal(cardinality_of(SP), 1)*]] $\rightarrow$ $\sim$ *equal(Tree,1)*] }

If we compare C with the postcondition, we can notice that the variable *RETURN* matches the predicate *equal*(*Tree*,1).

In the analogical unification process, we expand the substitutions in conventional unification algorithms (in which only variables are substituted) by including those for predicate heads (e.g., substituting *weighted_graph* for *airline*). Given two expressions to be unified, first, the numbers of predicates in them should be the same. More precisely, the number of relational predicates should be the same, and the number of method predicates should also be the same. Once the above requirements are met, for each possible pairing, the process of analogical unification can be applied. An analogical unification succeeds if we can find a set of substitutions (of predicate heads and variables) which is consistent throughout the unification process. By modifying the unification algorithm in [Nils80], we can derive a recursive algorithm which analogically unifies two expressions:

Algorithm 1. Analogical Unification Of Two Expressions
INPUT: Two expressions E_1 and E_2.
OUTPUT: A list of substitutions.

1) If E_1 and E_2 are variables, return E_1/E_2.
2) If E_1 is *RETURN* and E_2 is a constant or a predicate, return E_2/E_1.
   If E_2 is *RETURN* and E_1 is a constant or a predicate, return E_1/E_2.
3) If E_1 is a variable and E_2 is a constant, return E_2/E_1.
4) If E_2 is a variable and E_1 is a constant, return E_1/E_2.
5) If E_1 is a predicate head, a connective or a constant, do the following:
   5.1) If E_1 equals E_2, return *NIL*.
   5.2) If E_1 and E_2 are predicate heads for object classes, compare the attributes and their domains. If they are comparable, return E_1/E_2. Otherwise, return *FAIL*.
   5.3) If E_1 and E_2 are predicate heads for methods, analogically unify their preconditions and their postconditions. If the unifications succeed, return E_1/E_2. Otherwise, return *FAIL*.
   5.4) Otherwise, return *FAIL*
6) If E_2 is a predicate head, a connective or a constant, return *FAIL*.
7) Set $FIRST_1$ = the first element of E_1 and $REST_1$ = the rest of E_1.
   Set $FIRST_2$ = the first element of E_2 and $REST_2$ = the rest of E_2.
   Set S_{FIRST} = the result of analogical unification of $FIRST_1$ and $FIRST_2$.
   If S_{FIRST} is *FAIL*, return *FAIL*.
8) Set $MODIFIED_1$ = the result of applying S_{FIRST} to $REST_1$.
   Set $MODIFIED_2$ = the result of applying S_{FIRST} to $REST_2$.
   Set S_{REST} = the result of analogical unification of $MODIFIED_1$ and $MODIFIED_2$.
   If S_{REST} is *FAIL*, return *FAIL*.
   Otherwise, return the composition of S_{FIRST} and S_{REST}.

□

Once a program analogically unifies an algorithm, a method which implements the algorithm can be used for the program. In order to include a method into a program, objects used in the program should be restructured before they can be used as the arguments for the method. For example, the attribute names for all object instances in class *airline* (e.g., *CITIES* and *FLIGHTS*) should be renamed according to the attribute names in the class *weighted_graph* (e.g., V and E) in order to be used as the arguments for the method *shortest_path*. After the method is evaluated, the results need to be interpreted properly (e.g., a shortest path should be interpreted as a cheapest connection). This substitution and interpretation should be based on the

unifier set derived from the analogical unification process. Given a unifier set, the change of attribute names and interpretations can be done mechanically.

Example 1. For the problem of finding the cheapest connection in an airline reservation system, we can find an analogical match with the method *shortest_path* in class *graph*. In the match we have two sets of substitutions as follows:

Substitutions for Variables =

{ *F/G, CITIES/V, FLIGHTS/E,* c_1/V1 c_2/V2, C_SET/P_SET, Y/X, SC/SP, FY/WX, FSC/FSP }

Substitutions for Predicate Heads =

{ *airline/weighted_graph, all_connection/all_path, sum_of_fare/sum_of_weight* }

In order to call the method *shortest_path* in the program, the attribute names in class *airline* should be changed temporarily according to the substitution set (in this example, the variable names happen to be the same as the corresponding attribute names). The change of attribute names can be listed as follows:

$CITIES \rightarrow V$
$FLIGHTS \rightarrow E$
$FARE \rightarrow WEIGHT$

Once a solution *SP* (which is a list of nodes) is derived, the above modification should be applied reversely and the solution should be interpreted as a list of *CITIES*

□

In the above example, because the preconditions of the program and the method are {*true*}, we only need to unify their postconditions in order to substitute the method for the program. However, if a part of a program is tried to be unified, the precondition of the part may include some assertions which are not relevant to the part but to the status of the program as a result of executing the previous portion of the program. Suppose that a program consists of two parts A and B, where part A and part B are connected serially so that the postcondition of part A and the precondition of part B are the same (i.e., $Q_A = P_B$). In this case, a method M can be substituted for part B if the following conditions are satisfied:

(a) P_B and Q_B can be decomposed into

$$P_B = C \wedge P$$
$$Q_B = D \wedge Q$$

where C implies D, i.e., $C \rightarrow D$.

(b) P can analogically unify the precondition of the method, and Q can analogically unify the postcondition of the method, i.e.,

P *analogically unifies* P_M
Q *analogically unifies* Q_M

As an example, consider a program which sorts a set of *flights* after it computes the sum of the cost for all the *flights* in the set. If we consider the program segment which computes the sum as part A and the other program segment as part B, we can have the following assertions (here we assume that part A assigns the result to *SUM*, and part B sorts in ascending order the *flights* from the set *F_SET* according to their *COST* and sets the result to *SORTED_LIST*):

P_A = { *true* }
Q_A = { *equal(SUM, summation(COST, F_SET))* }

P_A = { *equal(SUM, summation(COST, F_SET))* }

P_A = { *equal(SUM, summation(COST, F_SET))* ∧ *[(∀ X) (∀ Y) [member_of(X, SORTED_LIST)* ∧ *member_of(Y, SORTED_LIST)* ∧ *lessthan(X.COST, Y.COST)* → *precede(X, Y, SORTED_LIST)]]* }

Suppose that we have a method M, which sorts a set of *weighted_edges*, with the following precondition and postcondition:

P_M = { *true* }

P_M = { *(∀ U) (∀ V) [member_of(U, LIST)* ∧ *member_of(V, LIST)* ∧ *lessthan(U.W, V.W)* → *precede(U, V, LIST)]* }

Using the notations in conditions (a) and (b) above, we can derive the following:

C = { *equal(SUM, summation(COST, F_SET))* }

D = { *equal(SUM, summation(COST, F_SET))* }

P = { *true* }

Q = { *(∀ X) (∀ Y) [member_of(X, SORTED_LIST)* ∧ *member_of(Y, SORTED_LIST)* ∧ *lessthan(X.COST, Y.COST)* → *precede(X, Y, SORTED_LIST)]* }

Because all the conditions in (a) and (b) are satisfied, i.e.,

$C \rightarrow D$,
P analogically unifies P_M, and
Q analogically unifies Q_M,

we can substitute the method M for part B with proper instantiations.

Once we have a successful unification between two object classes or between two methods, we can save the relationship and reuse it later. The collection of previously proved relationships can be considered as a *library of substitutions*. If two methods are saved in the library of substitutions, the unification process can be much easier because we only need to unify the arguments without unifying their preconditions and postconditions. For example, after successfully unifying a program with the method *shortest_path* in the above example, we can save the following relationship in the library of substitutions:

weighted_graph	↔*airline*
all_path	↔*all_connection*
sum_of_weight	↔*sum_of_fare*

The overall process of analogical unification can be described as a set of meta rules such as the following, where P_X and Q_X represent the precondition and the postcondition, respectively, of a method or a program X, and we assume that the predicates in a rule are evaluated from left to right.

If *unifiable(A, B, S)*
Then *compatible(A, B, S)*
where *unifiable(A, B, S):* which is true if two expressions A and B are unifiable with a unifier S by the conventional unification process.

5. Global Search Strategies

Given a program specified in a flowchart language, the best thing to do is to replace the whole program by an efficient algorithm. However, in a real situation, it is not always possible to find an algorithm which exactly matches a given program. If we fail to transform the entire program, then we can try to substitute parts of the program by good algorithms. In this section, some global search strategies are discussed, which can be encoded as meta rules. The global search strategies include the following:

(a) *The overall search method:*
By the rule of composition and the rule of iteration, in a program written in a flowchart language, two or more nodes can be composed into one. The process of successive compositions forms a tree, where a set of children is composed into their parent recursively until the whole program is represented by a root node. Each node in the tree represents a part of a program to be substituted by a proper algorithm. If no proper algorithm is found for a node, it can be composed with adjacent nodes or decomposed into details and any proper substitution is searched again. Considering this, the overall search process for finding an optimal transformation can be a top-down search, a bottom-up search, or various combinations of both.

(b) *Preference in selecting rules:*
When more than one object-level rule is applicable at a time, the most promising rule should be selected in order to optimize the overall transformation. The preferences among the rules can be determined by comparing the algorithms which are substituted by the rules. In general, algorithms can be compared by their orders of complexities. Two orders of complexity can be compared if they are expressed by means of one variable (e.g., n^2 and n^3). However, if the orders of complexity are defined in terms of multiple variables (e.g., $|E|^2$ and $|E|\,|V|$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in a graph), we need some information about the object classes (e.g., the cardinalities of E and V) in order to compare these.

(c) *Policies to solve ambiguities:*
When a program is non-deterministic, a transformed program can produce results which are correct but different from those generated by the original program. For example, the algorithm *topological_ordering* can produce different orderings according to implementations. In this situation, only with logical assertions, it is not always possible to assure that the new program generates the same results as the original program. One way to resolve this problem is to get a confirmation from the user. The program can be transformed with user's grant; otherwise, only subparts of the program will be searched for transformation.

6. Related Work

Related work to the program transformation and optimization described in this paper can be classified into three categories: program transformation, program reuse, and knowledge-based editors.

Program transformation includes predefined transformations (e.g., rewriting rules) and program constructions from a high level nonexecutable specifications to a low level executable form. Existing transformation systems can be divided into two classes: those that perform transformations automatically and those which are guided by users. The CIP project [CIP84] [BMPP89] focused on correctness-preserving and source-to-source program transformation at different levels of abstraction. The development process is guided by the programmer who has to choose appropriate transformation rules. The user guidance accomplishes the creative part in the

development process. DEDALUS [MaWa79] and KBSA [PrSm88] attempted to automate the transformation selection process. DEDALUS was able to create a program, a correctness proof, and a proof of termination for programs of a limited scope. DEDALUS selects candidate rules by pattern-directed invocations and applies those rules sequentially. KBSA focused on automatic algorithm design, deductive inference, finite differencing, and data structure selection. Given a problem description, KBSA generates an optimal program through correctness-preserving transformations. One of the problems for existing automatic program transformation systems is the lack of driving force of a design process. Even though some search approaches such as cost functions and efficient search methods have been employed, global strategies have not been integrated effectively. In KBPTS, global strategies were implemented as meta rules, in which control knowledge can be more clearly represented and executed [Cavi89] [OSGR89] [Past89].

Software reuse appears in two levels of abstraction: reuse at the code level and reuse at the specification level [Dill88]. While code-level reuse involves modifying existing code [PrFr87], specification-level reuse is based on an external, often formal, program specification. Existing methodologies include program transformation [Chea84] [BoMu84] and software components catalogue [WoSo88]. Program transformations are used to refine an abstract program defined in a very high level language into a program written in a target language. Software components catalogue requires the ability to match users' requests onto descriptions of software components which satisfy these requests. The main barriers to software reuse have been pointed out as follows:

(a) It is difficult to develop an enough set of generalized components for potential reuse [WoSo88].

(b) Even with a catalogue of reusable components, the matching processes are not effective enough [Dill88].

In STA, a collection of abstract algorithms replaces the software components catalogue, where abstract algorithms form a smaller set than reusable software components. The search method in KBPTS is an extended unification process, which is general enough to find existing solutions.

Simple program editors have been extended to be more powerful ones. Some incorporate an understanding of the syntactic structure of the program being edited [MeFe81] [TeRe81]. This makes it possible to support operations based on the parse tree of a program (e.g., inserting, deleting, and moving between nodes in the parse tree). Syntax-based editors also ensure the syntactic correctness of the program being edited. KBEmacs [Wate85] extended program editors further by including an understanding of the algorithm structure of the program. By comparing the algorithm structures with programming cliches, which are standard models of solving programming problems, KBEmacs can intelligently assist programmers. KBEmacs assists programmers to construct programs more rapidly and more reliably by combining or modifying existing algorithmic cliches. The idea of using algorithmic cliches is similar to that of using abstract algorithms in KBPTS. One difference is that cliches are domain dependent reusable components while abstract algorithms are general ones which can be applied to problems in various domains.

7. Conclusion

In this paper, we have discussed a knowledge-based program transformation and optimization system. Given a program, it can be transformed into a (or a set of) well-designed abstract algorithm by comparing their preconditions and their postconditions. To compare two conditions, the conventional unification algorithm has been modified into an *analogical unification* algorithm, where two predicates with different heads can be unified if the corresponding classes or methods are compatible with each other. When we allow transformations of subparts of a program, the search space could be increased enormously. In order to control the search space more clearly, a set of global search strategies is proposed.

Parts of the transformation process have been implemented and some examples are demonstrated in [Yoo90]. From the experimentation, we have learned that a logical description of a program (by a precondition and a postcondition) can easily include errors if we do not have some rigid rules. Also, if the description of a program includes some assertions which are not relevant to the functionality of the program, it is difficult to search for an appropriate algorithm. An example of the irrelevant assertions is the description of local variables in a program. To be more practical and general, it is essential to develop a standard method which is applicable to logical descriptions of algorithms and programs.

References

[Bars87] Barstow, D., "Artificial Intelligence and Software Engineering," *9th International Conference on Software Engineering,* 1987, pp. 200-211.

[BeKi89] Bertino, E., and Kim, W., "Indexing Techniques for Queries on Nested Objects," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 1, No.2, June, 1989, pp. 196-214.

[BoMu76] Bondy, J., and Murty, U., *Graph Theory with Application,* Macmillan Press Ltd., Great Britain, 1976.

[BoMu84] Boyle, J., and Muralidharan, M., "Program Reusability through Program Transformation," *IEEE Trans. on Software Engineering,* Vol. SE-10, No. 5, Sept. 1984, pp. 574-588.

[BMPP89] Bauer, F., Moller, B., Partsch, H., and Pepper, P., "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming," *IEEE Trans. on Software Engineering,* Vol. 15, No. 2, Feb. 1989, pp. 165-180.

[Carb81] Carbonell, J., "A Computational Model of Analogical Problem Solving," *Proceedings of IJCAI,* 1981, pp. 147-152.

[Cavi89] Caviedes, J., et al., "A Meta-Knowledge Architecture for Planning and Explanation In Repair Domains." in *Knowledge-Based System Diagnosis, Supervision, and Control,* Tzafestas, S. ed., Plenum Press, New York, 1989.

[Chea84] Cheatham, T., " Program Reusability Through Program Transformation," *IEEE Trans. Software Engineering,* Sept. 1984.

[CIP84] CIP Language Group, *Lecture Notes in Computer Science. Volume I: The Munich project CIP,* Spring-Verlag, 1984.

[DaBu84] Davis, R., and Buchanan, B., "Meta-level Knowledge," in *Rule-based Expert Systems,* Buchanan, B., and Shortliffe, E. eds, Addison-Wesley, Reading, Massachusetts, 1984.

[Ders86] Dershowitz, N., "Programming by Analogy," in *Machine Learning: An Artificial Intelligence Approach, Vol. 2,* Michalski, R., Carbonell, J., and Mitchell, T. eds, Morgan Kaufmann Publishers, Inc., 1986, pp. 395-423.

[Dill88] Dillistone, B., "Configuration management within an IPSE and its implications for software re-use," in *Software Engineering Environments,* Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[Floy67] Floyd, R., "Assigning meanings to programs," *Proc. Amer. Math. Soc. Symposia in Applied Mathematics,* Vol. 19, 1967, pp 19-31.

[FoSL83] Foderaro, J., Sklower, K., and Layer, K., *The Franz Lisp Manual* University of California, 1983.

[Hoar69] Hoare, C., " An axiomatic basis for computer programming," *Comm. ACM,* Vol. 12, No. 10, Oct. 1969, pp 576-580.

[MaWa79] Manna, Z., and Waldinger, R., "Synthesis: Dreams ⇒ Programs," *IEEE Trans. on Software Engineering,* Vol. SE-5, No. 4, July 1979, pp. 294-328.

[MeFe81] Medina-Mora, R., and Feiler, P., "An incremental programming environment," *IEEE Trans. on Software Engineering,* Vol. SE-7, Sept. 1981.

[Nils80] Nilsson, N., *Principles of Artificial Intelligence,* Springer-Verlag, 1980.

[OSGR89] Oussalah, C., Santucci, J., Giambiasi, N., and Roux, P., "Expert system based on multi-view/multi-level model approach for test pattern generation," in *Knowledge-Based System Diagnosis, Supervision, and Control,* Tzafestas, S. ed., Plenum Press, New York, 1989.

[Past89] Pastre D., "MUSCADET: An Automatic Theorem Proving System Using Knowledge and Metaknowledge in Mathematics," *Artificial Intelligence,* Vol. 38, 1989, pp 257-318.

[PrFr87] Prieto-Diaz, R., and Freeman, P., "Classifying software for reusability," *IEEE Software,* Vol. 4, No. 1, January 1987, pp 6-16.

[PrSm88] Pressburger, T., and Smith, D., "Knowledge-based software development tools," in *Software Engineering Environments,* Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[Robi65] Robinson, J., "A Machine-Oriented Logic Based on the Resolution Principle," *JACM,* Vol. 12, No. 1, January 1965, pp. 23-41.

[Sheu88] Sheu, P.C-Y., "Describing Semantic Databases in Logic," *Journal of Systems and Software,* 1988.

[Sheu90] Sheu, P.C-Y., "OASIS - An Object-oriented And Symbolic Information System," *Proc. First Int'l Conf. on System Integration,* April, 1990.

[Shoo83] Shooman, M., *Software Engineering: Design, Reliability and Management,* McGraw-Hill Book Co., 1983.

[ShYo90] Sheu, P. C-Y., and Yoo, S.B., "A Knowledge-Based Software Environment (KBSE) for Designing Concurrent Processes," *International Journal of Human-Computer Interactions,* Vol. 1, No. 2, pp. 161-185, 1990.

[TeRe81] Teitelbaum, T., and Reps, T., "The Cornell program synthesizer: A syntax-directed programming environment," *Communication of ACM,* Vol. 24, No. 9, Sept. 1981.

[Wate85] Waters, R., "The Programmer's Apprentice: A Session with KBSmacs," *IEEE Trans. on Software Engineering,* Vol. SE-11, No. 11, November 1985.

[Wile84] Wilenski, R., *LISPcraft,* W.W.Norton & Company, Inc., New York, N.Y., 1984.

[WoSo88] Wood, M., and Sommerville, I., "A knowledge-based software components catalogue," in *Software Engineering Environments,* Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[Yoo90] Yoo, S., "Integrated Process Management In A Parallel Database Programming Environment," Ph.D. Thesis, School of Electrical Engineering, Purdue University, May 1990.

[Zani84] Zanilo, C., "Object-base Logic Programming," *Symp. on Logic Programming,* 1984.