TR-135

Retrieval of Software Module Functions Using First-order Predicate Logical Formulae

by
H. Yoshida, H. Kato and M. Sugimoto
(Fujitsu Ltd.)

September, 1985

RETRIEVAL OF SOFTWARE MODULE FUNCTIONS USING FIRST-ORDER PREDICATE LOGICAL FORMULAE

H. Yoshida, H. Kato, and M. Sugimoto

FUJITSU LIMITED
1015 Kamikodanaka Nakahara-ku, Kawasaki, Japan

ABSTRACT

This paper introduces a method to retrieve software modules from a module library in order to reuse them for new software. It is effective for a programming environment in which specifications of software modules are formalized using first-order predicate logical formulae. This method uses resolution and heuristics to determine reusability of current modules in the library. A prototype system has been developed using C-Prolog on a VAX11/780.

1. PREFACE

In logic programming, a programming language has its logical basis in Horn clauses which are subsets of first-order predicate logical formulae. A language for specifications of logic programs will be somewhat close to logical expressions. For this reason, first-order predicate logical formulae can be taken as one of the candidates for the specification description language. However, first-order predicate logical formulae are both difficult to write and read, and thus require a user interface. The TELL system (Enomoto et al. 1984) currently being researched by the Tokyo Institute of Technology is an attempt to integrate a natural-language interface into formal specification descriptions written in first-order predicate logical formulae. In other words, the specification description language TELL/NSL, which is the core of the TELL system, is a natural language (English) with limited syntax, and software specifications written in this language can be translated unambiguously into first-order predicate logical formulae through syntactic analyses.

Well-written software specifications not only facilitate maintenance, but also make it easier to reuse old software. If useful software is stored in a library and can be retrieved easily, it can be reused, which improves productivity significantly. Although automatic retrieval is desirable, simply matching characters is not a practical method of retrieval. A retrieval method based on software functions is required. It is important to grasp the "semantics of specifications" as intended by the designer for semantical matching. This paper discusses a method for retrieving software module functions in a programming environment, such as the TELL system, that can formalize semantics of specifications using first-order predicate logical formulae. This method uses resolution and heuristics to match first-order predicate logical formulae. This paper briefly introduces specification description language TELL/NSL in Section 2. It presents an overview, and the algorithm of the method in Sections 3 and 4, and reports on a prototype system developed using C-Prolog on a VAX11/780 in Section 5.

2. SPECIFICATION DESCRIPTION LANGUAGE TELL/NSL

The four types of modules that can be described using TELL/NSL are functional definition, equivalent to a predicate or function, class definition, equivalent to an abstract data type, action definition, equivalent to a parallel process, and dynamic class definition, equivalent to shared data between processes. This retrieval method currently applies only to functional definition.

Figure 1 is an example of a top level module definition for solving the eight queens puzzle described in TELL/NSL. This specification can be translated into Formula (1) below.

Arrangement X is an <u>eight queens solution</u>

  means that

    1) Eight queens are placed in X.

    2) No queen is checking any other queen in X.

end eight queens solution.

Fig. 1 A sample specification (by TELL/NSL)

$$\forall c\,[\text{eight\_queens'\_solution}(c) \equiv \exists S\,[|S|=8 \wedge \forall q\,[q \in S \equiv \text{placed}(q,c)]] \wedge \sim\exists q1 \exists q2\,[q1 \neq q2 \wedge \text{checking}(q1,q2,c)]] \quad (1)$$

TELL/NSL is a pseudo-natural language which can be translated into first-order predicate logical formulae. It is characterized by natural stepwise refinement through lexical decomposition. That is, one word corresponds to each module, and the specification description of each module explains the meaning of the word in natural language. The words in the explanatory statement provide further explanations as auxiliary modules which are subcontractors of the original module. Thus, the module structure and the inter-module interface of the entire software are determined naturally based on the explanatory statement in natural language. The specifications of each module define the logical association with the auxiliary modules and are translated into relatively simple logical formulae. Therefore, a sufficient response speed can be expected even if the resolution principle is used for those logical formulae. Since there is a tendency to generate many small modules, reusing modules by functional retrieval would be highly effective.

3. FUNCTIONAL RETRIEVAL

A functional retrieval system using this method is used, for example, in the following sequence:

(1) The specifications of individual modules using TELL/NSL, their translated logical formulae, and programs that have been verified to satisfy the logical formulae are stored in a library.

(2) Anyone can then use TELL/NSL to develop a new module by describing the specification of the module.

(3) A syntactic analysis is then performed on the specification of the new module, which is then translated into a first-order predicate logical formula and input into the functional retrieval system.

(4) The functional retrieval system compares the input logical formula with the formulae of the modules in the library to determine whether they are reusable.

(5) The developer processes the program of a module determined to be reusable and obtains a program of the new module.

Since this method uses the resolution principle to determine the equivalence of logical formulae, a specifications is determined to be reusable if it is logically equivalent to the input specification, regardless of differences in characters or expressions. For example, module eight_queen_puzzle with a specification that can be translated as shown in Formula (2) below is reusable as module eight_queens_solution shown in Figure 1.

$$\begin{aligned}&\forall c\ [\text{eight\_queens\_puzzle}(c) \equiv \\ &\qquad \forall q1 \forall q2\ [\text{checking}(q1,q2,c) \Rightarrow q1=q2] \\ &\qquad \wedge \exists S\ [|S|=8 \wedge \forall q\ [q \in S \equiv \text{placed}(q,c)]]] \end{aligned} \quad (2)$$

In reality, however, modules with equivalent functions are seldom stored in the library. To use the functional retrieval system effectively, modules that are "similar" to some extent must also be retrievable. The problem is under what condition the two specifications should be considered similar. Because the main purpose of the functional retrieval system is to promote reuse of programs, it must be able to obtain the desired program easily, merely by processing the programs of the retrieved modules. Therefore, the functional retrieval system must not only to answer "similar," but also provide guidelines as to what part of the program text should be processed and how. The method tentatively checks whether each of the following three cases and any of their combinations apply, to determine whether two specifications are "similar":

(a) Difference in the order of parameters

Two specifications are equivalent except for the order of formal parameters. For example, modules successor and predecessor having specifications that are translated as shown in Formulae (3) and (4), respectively, are logically equivalent if the two formal parameters are exchanged.

$$\forall x \forall y\ [\text{successor}(x,y) \equiv x=\text{increment}(y)] \quad (3)$$

$$\forall z \forall w\ [\text{predecessor}(z,w) \equiv \text{increment}(z)=w] \quad (4)$$

Therefore, if Formula (3) is input, the functional retrieval system will return Formula (5), provided that predecessor is stored in the library.

$$\forall x \forall y\ [\text{successor}(x,y) \equiv \text{predecessor}(y,x)] \quad (5)$$

By taking the program text of module predecessor out of the library and by replacing all occurrences of the first parameter z with y and all of the second parameter w with x, the user can obtain the program of the new module successor.

(b) Turning a parameter into a constant

Two specifications will be equivalent if some constant is given as an actual parameter. For example, module n_queens_solution having the specifications that can be translated as shown in Formula (6) will be reusable as module eight_queens_solution shown in Figure 1 if constant 8 is given as the first parameter.

$$\forall n \forall c\ [n\_queens'\_solution(n,c) \equiv \exists S\ [|S|=n \wedge \forall q\ [q \in S \equiv placed(q,c)]] \wedge \sim\exists q1 \exists q2\ [q1 \neq q2 \wedge checking(q1,q2,c)]] \quad (6)$$

Therefore, if Formula (1) is input, the functional retrieval system will return Formula (7).

$$\forall c\ [eight\_queens'\_solution(c) \equiv n\_queens'\_solution(8,c)] \quad (7)$$

By taking the program text of module n_queens_solution out of the library and by replacing all occurrences of the first parameter n with 8, the user obtains the program of the new module eight_queens_solution.

(c) Difference in auxiliary modules

Two specifications will be equivalent if the subcontracting auxiliary modules are replaced with similar ones. For example, in module sort and module generate_test having the specifications that can be translated as shown in Formula (8) and Formula (9), respectively, the original modules are equivalent assuming that auxiliary modules parmutation and generated, and sorted and tested are logically equivalent.

$$\forall x \forall y\ [sort(x,y) \equiv permutation(x,y) \wedge sorted(y)] \quad (8)$$

$$\forall z \forall w\ [generate\_test(z,w) \equiv generated(z,w) \wedge tested(w)] \quad (9)$$

Therefore, if Formula (8) is input, the functional retrieval system will return Formula (10), and Formulae (11) and (12) which are assumed equivalences of auxiliary modules.

$$\forall x \forall y\ [sort(x,y) \equiv generate\_test(x,y)] \quad (10)$$

$$\forall x \forall y\ [permutation(x,y) \equiv generated(x,y)] \quad (11)$$

$$\forall x\ [sorted(x) \equiv tested(x)] \quad (12)$$

By taking the program text of module generate_test out of the library and by replacing all occurrences of the auxiliary module names generated and tested with permutation and sorted, respectively, the user obtains the program of the new module sort. It should be noted that this method does not verify the assumptions in Formulae (11) and (12). Therefore, modules having completely different functions may be retrieved by this method. However, as shown above, the program of module generate_test can easily be reused regardless of whether Formulae (11) and (12) are satisfied. Auxiliary modules parmutation and sorted of the new module sort can be used as is if they already exist as modules. If they are new modules, specifications are defined for them and reusable modules are functionally retrieved from the library as for sort, in which case the retrieved modules need not be generated or tested.

4. PRINCIPLES OF RETRIEVAL

Parameters of each module are represented by universally quantified logical variables in a logical formula which is a translation of specifications. Therefore, the similarities shown in Sections (a) and (b) of the preceding section can be determined, because their logical variables or their logical variables and constants are unified during the process of verification of equivalence of logical formulae, by the resolution principle. The determination of the similarity shown in Section (c), on the other hand, is a high order unification, and is realized by heuristics.

The procedures and heuristics used are explained below.

(a) Terminology and notation

Predicate names are represented by lower case alphabetics, such as p, f, and g, while the ordinary logical formulae are represented by upper case alphabetics, such as F and G. The set of parameters X1, X2, ..., Xn of predicate p is represented by "$\underline{X}$" as a tuple. Therefore, call of p is represented by "$p(\underline{X})$". The operation for properly rearranging the order of tuple elements is called permutation. Tuple $\underline{X}$ rearranged by parmutation π is represented by "$\underline{X}\pi$". Tuple <X1, X2, ..., Xi, Y1, Y2, ..., Yj> for the two tuples $\underline{X}$=<X1, X2, ..., Xi> and $\underline{Y}$=<Y1, Y2, ..., Yj> is called concatenation of $\underline{X}$ and $\underline{Y}$, and is represented by "$\underline{X} + \underline{Y}$".

Call of a predicate and its negation is called a literal. In particular, call $p(\underline{X})$ of predicate p and its negation $\sim P(\underline{X})$ are called positive literal of p and negative literal of p, respectively.

A clause is a set of literals. A clause consisting of only one literal is called a unit clause; in particular, a clause consisting of only one literal of predicate p is called the unit clause of p.

(b) Procedures

Hereafter, the new module to be retrieved will be named f, and the module with which its equivalence is to be verified will be named g. Modules f and g are defined in Formulae (13) and (14).

$$\forall \underline{X}\, [f(\underline{X}) \equiv F(\underline{X})] \quad (13)$$

$$\forall \underline{Y}\, [g(\underline{Y}) \equiv G(\underline{Y})] \quad (14)$$

In the first phase, it is determined whether functions of module g satisfy those of module f. Functions are verified if a parameter tuple $\underline{Z}$ of module g that satisfies Formula (15) can be found for a parameter tuple $\underline{X}$ of module f.

$$\forall \underline{X}\, [g(\underline{Z}) \Rightarrow f(\underline{X})] \quad (15)$$

In this case, since Formula (15) to be verified is unknown, the normal resolution principle cannot be applied as is. In this retrieval method, therefore, parameter tuple $\underline{X}$ of module f is fixed to obtain Formula (18) as the unit clause of module g, using Formulae (16) and (17) as axioms.

$\sim F(\underline{X})$ (16)

$\forall \underline{Y}\ [G(\underline{Y}) \Rightarrow g(\underline{Y})]$ (17)

$\sim g(\underline{Z})$ (18)

If Formula (18) can be resolved, formula (19) is considered to be satisfied, and Formula (15) is considered to have been verified by Formulae (19) and (13).

$\sim F(\underline{X}) \Rightarrow \sim g(\underline{Z})$ (19)

At this time, tuple $\underline{Z}$ is normally a permutation of tuple $\underline{X}$ (or a concatenation of tuple $\underline{X}$ with a tuple consisting of several constants).

With Formula (15) alone, module g may be a partial solution of module f. Therefore, Formula (20) which is the opposite of Formula (15) is also verified in the second phase.

$\forall \underline{X}[f(\underline{X}) \Rightarrow g(Z)]$ (20)

Formula (21) can be resolved from Formulae (15) and (20). Thus, module g which is reusable as f is discovered.

$\forall \underline{X}\ [f(\underline{X}) \equiv g(\underline{Z})]$ (21)

(c) Heuristics

(1) Forced factoring of literals of g

This is a unique heuristic in the first phase of this method, which is used to improve the efficiency of the resolution principle. That is, if the resolved clause contains multiple negative literals of g and if they can be factored, an attempt is made to forcibly factor them so that the resolved clause contains only one negative literal of g. If the literals cannot be factored, the resolved clause is deleted, because the clause to be finally resolved in the first phase must not be an empty clause. That is, it must be a unit clause of g, such as Formula (18).

(2) Determination of similarity of auxiliary modules

This is a heuristic used to find a set to advance the inference, where the number of parameters in auxiliary module h of f and auxiliary module k of g is assumed to be equal and h and k are equivalent. In the first phase, the two clauses C1 and C2 that satisfy the following conditions are searched for among the resolved clauses:

- C1 contains positive literal $h(\underline{V})$ of h and C2 contains negative literal $\sim k(\underline{W})$ of k. Or, C1 contains negative literal $\sim h(\underline{V})$ of h and C2 contains positive literal $k(\underline{W})$ of k.

- There is a permutation of the tuple, the class of the corresponding elements of $\underline{V}\pi$ and $\underline{W}$ is equal, and each of $\underline{V}\pi$ and $\underline{W}$ has a most general unifier σ.

- If C1 contains negative literal $\sim g(\underline{Y})$ of g, and C2 contains $\sim g(\underline{Z})$ of g, $\underline{Y}\sigma$ and $\underline{Z}\sigma$ are unifiable.

If the above conditions are satisfied, and if h and k (when parameters are rearranged by permutation π) can be equivalent, a new clause can be resolved from C1 and C2. Even if multiple negative literals of g appear in the clause, the new clause will not be excluded, by the heuristics of (1) above. Verification continues with Formula (22) added to the axiom.

$$\forall \underline{V}\ [h(\underline{V}) \equiv k(\underline{V}\pi)] \quad (22)$$

Formula (22) is also used as one of the axioms in the second phase.

(3) Handling of recursive definitions

Strictly speaking, to determine whether two recursively defined modules are equivalent, the mathematical induction on the data structure of the two must be used. However, it is difficult to do so automatically with a computer. And one can often decide that the two modules appear to be equivalent without using such a precise method. What must be noted here is how the recursive calls appears in the body of the modules. The decision based on the style of calling the subcontractors is the same as that of heuristic (2) above. So the method similar to that of (2) can be used to determine the equivalence of recursive modules.

In the first phase, the two clauses C1 and C2 that satisfy the following conditions are searched for among the resolved clauses:

- C1 consists of only negative literal $\sim f(\underline{V})$ of f and negative literal $\sim g(\underline{Z})$ of g.
- C2 consists of only positive literal $g(\underline{W})$ of g and negative literal $\sim g(\underline{Z})$ of g.
- Both tuple $\underline{A}$ consisting of only constants (including a tuple whose length is 0) and permutation π exist, $(\underline{V} + \underline{A})\pi$ and $\underline{W}$ have the most general unifier σ, and $(\underline{X} + \underline{A})$ and $\underline{Z}\sigma$ are equal.

When the above conditions are satisfied, the first phase terminates with Formula (23) considered to be satisfied.

$$\forall \underline{X}\ [g(\underline{Z}\sigma) \Rightarrow f(\underline{X})] \quad (23)$$

Recursive calls are processed in the same way in the second phase.

It will be shown that this heuristic is correct for a module recursively defined for the list structure. For simplicity, it is assumed that both f and g have two parameters. The first parameter is used for input and the second parameter is used for output, and they are defined by Formulae (24) and (25).

$$\forall x \forall y\ [f(x,y) \equiv F(x,y)] \quad (24)$$

$$\forall x \forall y\ [g(x,y) \equiv G(x,y)] \quad (25)$$

In the first phase, arguments x and y of module f are fixed to resolve the unit clause of module g using Formulae (26) and (27) as axioms, and heuristic (3) checks that a clause as shown in Formulae (28) and (29) is resolved.

$\sim F(x,y)$ (26)

$\forall v \forall w\ [G(v,w) \Leftrightarrow g(v,w)]$ (27)

$\sim f(cdr(x),y)\ v \sim g(x,y)$ (28)

$g(cdr(x),y)\ v \sim g(x,y)$ (29)

If the clauses of Formulae (28) and (29) are resolved, then Formulae (30) and (31) can be resolved from Formulae (24), (25), (26) and (27).

$\forall x \forall y\ [g(x,y) \wedge f(cdr(x),y) \Leftrightarrow f(x,y)]$ (30)

$\forall x \forall y\ [g(x,y) \Rightarrow g(cdr(x),y)\ v\ f(x,y)]$ (31)

Now, it will be verified that Formula (32) is satisfied by using the mathematical induction on a list structure.

$\forall x \forall y\ [g(x,y) \Rightarrow f(x,y)]$ (32)

(1) When x is nil

Since literal g(cdr(x),y) means "$\exists z[cdr(x,z) \wedge g(z,y)]$", it is false in this case, and Formula (33) can be resolved from Formula (31).

$\forall y\ [g(nil,y) \Rightarrow f(nil,y)]$ (33)

(2) When x is cdr(a)

Assuming Formula (34), Formula (35) can be resolved from Formulae (30) and (31).

$\forall y\ [g(cdr(a),y) \Rightarrow f(cdr(a),y)]$ (34)

$\forall y\ [g(a,y) \Rightarrow f(a,y)]$ (35)

5. PROTOTYPE SYSTEM AND EXAMPLE OF REUSE

A prototype of the functional retrieval system using this method has been realized using C-Prolog on a VAX11/780. The program consists of approximately 2000 lines, and whether logical formulae are equivalent is determined using ordered linear resolution (Chang et al. 1973). Of the heuristics discussed in Section 4, heuristics (1) and (3) are used as required, if they are applicable. Heuristic (2) is considered only when the resolution principle comes to a deadlock.

This prototype operates as one of the modules composing the program development environment (Sugimoto et al. 1984), whose specification description language is TELL/NSL. The prototype is also used as a subcontractor of another module called the semi-automatic synthesis system. The semi-automatic synthesis system is used interactively by the software designer. This system functions include specification description by TELL/NSL, translation of logical formulae by parser, retrieval of reusable modules by the functional retrieval system, reference and modification of the retrieved modules, and storing of the modified modules into the library.

Figure 2 is an example of synthesis of program on_the_same_column, which reuses the module on_the_same_row.

(a) New Specification (by TELL/NSL)

```
Queen Q1 and queen q2 are
             on the same column of arrangement x
  means that
  1) The x_coordinate of the position of q1 in x is
     the x_coordinate of the position of q2 in x.
end
```

(b) Translated Logical Formula of (a)

```
∀q1,q2,x [on_the_same_column [q1,q2,x] ≡
            x_coordinate [position [q1,x]]
             = x_coordinate [position [q2,x]]]
```

(c) Retrieved Module Name

```
on_the_same_row
```

(d) Translated Logical Formula of (c)

```
∀q1,q2,x [on_the_same_row [q1,q2,x] ≡
            y_coordinate [ position [ q1,x ]]
             = y_coordinate [position [q2,x]]]
```

(e) Assumption of the Equivalence

```
∀x [x_coordinate [ x ] ≡ y_coordinate [ x ]]
```

(f) Program of (c) (by Prolog)

```
on_the_same_row(Q1,Q2,X):  -
      position(Q1,X,P1),
      y_coordinate(P1,x),
      position(Q2,X,P2),
      y_coordinate(P2,X).
```

(g) Modified Program of (a)

```
on_the_same_column(Q1,Q2,X):  -
      positon(Q1,X,P1),
      x_coordinate(P1,Z),
      position(Q2,X,P2),
      x_coordinate(P2,Z),
```

Fig. 2 An example of program synthesis

(1) The user inputs specification (a) of the program to be obtained.

(2) The semi-automatic synthesis system converts specification (a) into logical formula (b) using the parser.

(3) Then, the semi-automatic synthesis system calls the functional retrieval system and retrieves similar modules.

(4) The functional retrieval system returns the name of the similar module (c), its logical formula (d), the logical formula of conditions under which the two agree (e), and the program (f).

(5) Viewing (f), the user determines it to be usable, modifies the program (f) based on (e), and obtains the program (g).

(6) The semi-automatic synthesis system stores the generated module into the library.

The user then repeats this cycle, if necessary, for further refinement (generating auxiliary modules).

6. CONCLUSION

This paper presented a method for retrieving reusable software modules by verifying the equivalence of first-order predicate logical formulae given as specifications. As long as the specifications are for programming environments that can be stipulated by first-order predicate logical formulae, this method can be applied, regardless of the implementation language.

This method is not, however, sufficient to retrieve all modules having specifications logically equivalent to those input by the user. It is also true that the specification_ of the retrieved modules are not always equal to new specifications. This is mainly because the auxiliary modules and the specifications of the data structure to be operated by them are not referenced when determining whether two modules are equivalent. For software development, however, it is desirable to determine the usable modules at the earliest possible stage to promote the reuse of modules, as well as to enable retrieval without complete detailed descriptions of the specifications. Therefore, the objective of this method is not to verify the logical equivalence including specifications of the modules (auxiliary predicates and data classes) being referenced, but to find similarities in their "reference formats." The method is based on the theory that if the reference formats in specifications are similar, the interfaces based on the programs that implement them are also similar and, therefore, can be easily used. In this method, the equivalence of specifications of auxiliary predicates is determined solely by how they are called and how parameters are given. The recursive equivalence is determined solely by the format of recursive call without using mathematical induction on the data structure. Using this method, a software designer can use any part of the reusable modules at each step of the stepwise refinement. The designer can also further refine unusable parts.

ACKNOWLEDGMENTS

This study was part of the fifth generation computer project. We would like to thank Institute for New Generation Computer Technology for giving us the opportunity to conduct this study.

We would also like to thank Professor Enomoto (current Superintendent of the International Information Society Institute), Assistant Professor Yonezaki, and Assistant Saeki of Tokyo Institute of Technology for their valuable guidance on TELL.

REFERENCES

Chang C, et al. Symbolic logic and mechanical theorem proving, Academic Press, U.S.A. 1973.

Enomoto H, et al. Natural language based software development system tell, ICOT TR-067, Tokyo, 1984.

Sugimoto M, et al. Design concept for software development consulation system, ICOT TR-071, Tokyo, 1984.