ICOT Technical Report: TR-153

TR-153

Logic-Based Retrieval and Reuse of Software Modules

by
Hideki Katoh, Hiroyuki Yoshida and Masakatsu Sugimoto
(Fujitsu Ltd.)

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

LOGIC-BASED RETRIEVAL AND REUSE OF SOFTWARE MODULES

Hideki KATOH, Hiroyuki YOSHIDA, and Masakatsu SUGIMOTO

FUJITSU LIMITED
Kawasaki, Japan

ABSTRACT

We have been developing an intelligent programming support system as one of R & D activities of the Japanese fifth generation computer systems project.

The system supports the entire software development process and has four features; (1) English-like specification language, which is mechanically translated into a set of first order predicate logic formulae, (2) logic-based retrieval of software modules from a module library, proving functional equivalence of required module and stored ones, using a resolution based theorem prover augmented by heuristic (meta) rules, (3) interactive program composition, using the retrieval to find reusable modules, (4) generation of module's explanatory sentences in the specification language.

This paper mainly describes functional retrieval and program composition.

1. Introduction

We have been developing an 'intelligent software development support system', as one of R & D activities of the Japanese fifth generation computer systems (FGCS) project, to increase the productivity, reliability, and maintainability of software based on logic programming languages. This paper describes the system's current status.

To achieve above goals, it is important to raise the level of the description language. A reasonable candidate is a specification language. The most friendly and easy-to-read specification language is natural language, which is, however, ambiguous. Our system uses an English-like language to describe software specifications. The semantics of the language are defined by first-order predicate logic, which has theoretical basis and is expected to suit logic programming languages well.

Making specification languages can be used practically needs automatic generation of runnable programs from specifications. It is, however, very difficult to synthesize entire programs from their specifications automatically, so the system uses another approach; a program is composed interactively based on functional retrieval and reuse of software modules in a module library.

2. Overview

The system supports the entire software development process and has four features:

(1) It uses an English-like specification description language, called TELL/NSL[2], which is designed to be readable for not only men but also machines.
(2) It uses an interactive program composition based on functional retrieval and reuse of software modules.
(3) It uses functional retrieval[3], from a module library, of software modules that logically satisfy a given specification.
(4) It generates explanations of Prolog programs in a natural-language-like style.

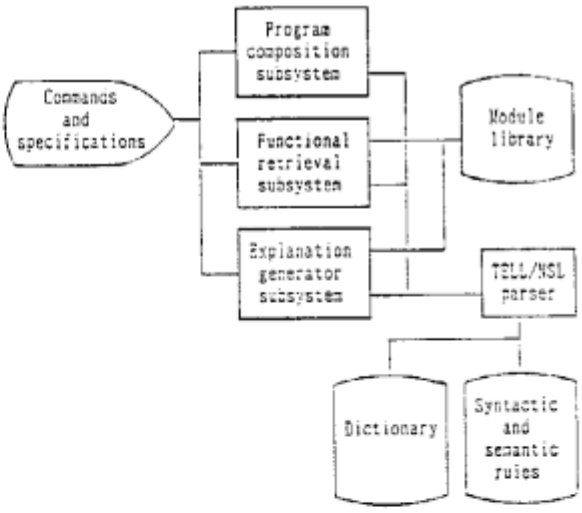

Fig.1 System configuration

The system consists of four subsystems and a module library, as shown in Fig.1. The parser analyzes specifications written in TELL/NSL and generates sets of logical formulae with the same meanings. The module library contains software modules; each module is divided into two parts: a specification, represented by logical formulae, and a program coded in Prolog. The module library is managed by the functional retriever.

Though there has been much research on automatic program synthesis, it is still far practical; our system uses another way, composing

/

programs from reusable software modules. This increases the productivity of programs, but it is difficult to find modules that will satisfy given specifications only after slight modification. The system finds reusable modules that have equivalent or similar specifications to required ones by using a theorem proving technique with heuristics.

Well-written software specifications not only facilitate maintenance, but also make it easier to reuse the software stored in the module library. If software in the module library can be retrieved easily it 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 programmer in one form or another for semantical matching. This section 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.

The system is implemented using C-Prolog on a VAX-11/780.

3. Specification Description Language TELL/NSL

The four types of modules that can be described using TELL/NSL are:

(1) Functional definition, equivalent to a predicate or function
(2) Class definition, equivalent to an abstract data type
(3) Action definition, equivalent to a parallel process
(4) Dynamic class definition, equivalent to shared data among processes

The system currently supports only functional definition.

Fig.2 shows, as an example, part of a specification of the eight queens puzzle using TELL/NSL and a corresponding set of logical formulae, which represents the meaning of the specification.

TELL/NSL is characterized by natural stepwise refinement through lexical decomposition. A word or phrase corresponds to a module, and the specification description of each module explains the meaning of the word or phrase in a natural-language-like form. 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 intermodule interface of the entire software are determined naturally based on the explanatory statement in TELL/NSL. The specifications of each module define the logical relationship to the auxiliary modules and are translated into relatively simple logical formulae.

The first sentence of the specification defines a predicate named eight_queens_solution, which has an parameter X that belongs to the class arrangement and is defined by the following two itemized sentences. The itemized sentences also include other predicates placed and checked, which are defined in a similar way as eight_queens_solution. This process repeats until predicates are defined by built-in predicates.

```
Arrangement X is an eight queens solution
 means that
  1) Eight queens are placed in X.
  2) No queen is checked by any other queen in X.
end eight queens solution;

Queen Q1 is checked by queen Q2 in arrangement X
 means that
  1) Q1 and Q2 are on the same row in X or
     Q1 and Q2 are on the same column in X or
     Q1 and Q2 are on the same diagonal in X.
end checked;
```

$$\forall x[\ \text{eight\_queens\_solution}(x) \equiv \neg\exists X0\exists X1[\ X0 \neq X1 \ \&\ \text{checked}(X0,X1,x)\ ]\ \&\ \neg\exists S[\forall q[q \in S \equiv \text{placed}(q,x)]\ \&\ |S|=8]\ ]$$

$$\forall x\forall q2\forall q1[\ \text{checked}(q1,q2,x) \equiv \text{on\_the\_same\_row}(q1,q2,x)\ |\ \text{on\_the\_same\_column}(q1,q2,x)\ |\ \text{on\_the\_same\_diagonal}(q1,q2,x)\ ]$$

Fig.2 Specification of the eight-queens puzzle using TELL/NSL

4. TELL/NSL Parser

The parser analyzes specifications written in TELL/NSL and generates in first-order logical formulae with the same meanings. It is implemented using a tool for building bottom-up parsers, called BUP[7]. The syntax and semantic rules of TELL/NSL represented by a DCG[8] are translated into a set of Prolog predicates by the BUP translator. These predicates construct the parser with a lexical analyzer, the BUP runtime routines, and other miscellaneous routines.

Strictly speaking, TELL/NSL is not a natural language, but a formal language, as it has well-defined syntactic and semantic rules. From the viewpoint of implementing its parser, however, it has characteristics of both natural and formal languages. This causes the following difficulties:

(1) It allows inflections of words, so a dictionary of irregular words is necessary.
(2) It has left recursive syntax rules, which does not enable the parser to use an effective top-down parsing algorithm.
(3) The above two problems come from natural language characteristics and, in turn, the formal language ones require some error check, which contradicts non-determinism. The parser has very little error checking, so it fails during analysis of illegal sentences, and provides no information to the user about the errors. This is big problem, because the users are forced learn the syntax precisely.

5. Functional Retrieval

5.1 Introduction

Most software module retrieval methods are based on textual matching of specifications or program codes, so they cannot find modules that are functionally equivalent to a required one but have a different form.

The principle of functional retrieval is to prove the equivalence of two specifications that are represented by first-order predicate logical formulae. It uses ordered linear resolution for this proof. The specifications of modules are translated into relatively simple logical formulae. Therefore, a sufficient response speed can be expected even if the resolution principle is used.

Simply proving equivalence is not sufficient when trying to retrieve software modules which are not equivalent, but which can be made equivalent with slight modification.

The prover is augmented by introducing heuristics, which are used when the process of proving equivalence sticks and adds some assumptions to axioms. Currently, the functional retriever has heuristics that change the order of parameters, make certain parameters constant, and/or rename subordinate predicates. These can be seen as a higher order unification.

It should be noted that the equivalence of two formulae includes the equivalence of their subordinate modules. For module retrieval, however, it is not necessary to prove complete equivalence, only partial or local equivalence. The program of the module found will generally be modified by the programmer for a specific use. The system trys to prove (under some assumptions) local equivalence, i.e., the formulae to be compared are the ones appeared in a module.

5.2 Examples

Since this method uses the resolution principle to determine the equivalence of logical formulae, a specification 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 (1) below is reusable as module eight_queens_solution shown in Fig.2.

$$\forall c[\text{eight\_queens\_puzzle}(c) \equiv \forall q1\forall q2[\text{checked}(q1,q2,c) \rightarrow q1=q2] \;\&\; \exists S[|S|=8 \;\&\; \forall q[q\in S \equiv \text{placed}(q,c)]]] \quad (1)$$

In reality, however, a module exactly equivalent to the desired one is 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 conditions should the two specifications 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 answer to similar, but must 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 translated as shown in Formulae (2) and (3), respectively, are logically equivalent if the two formal parameters are exchanged.

Therefore, if Formulae (2) is input, the functional retrieval system will return formula (4), provided that predecessor is stored in the library.

$$\forall x\forall y[\text{successor}(x,y) \equiv x=\text{increment}(y)] \quad (2)$$

$$\forall z\forall w[\text{predecessor}(z,w) \equiv \text{increment}(z)=w] \quad (3)$$

$$\forall x\forall y[\text{successor}(x,y) \equiv \text{predecessor}(y,x)] \quad (4)$$

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) 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 translated as shown in Formula (5) and Formula (6), respectively, the original modules are equivalent, assuming that auxiliary modules permutation and generated, and sorted and tested are logically equivalent.

$$\forall x\forall y[\text{sort}(x,y) \equiv \text{permutation}(x,y) \;\&\; \text{sorted}(y)] \quad (5)$$

$$\forall z\forall w[\text{generate\_test}(z,w) \equiv \text{generated}(z,w) \;\&\; \text{tested}(w)] \quad (6)$$

Therefore, if Formula (5) is input, the functional retriever will return Formula (7), and Formulae (8) and (9) which are assumed equivalences of auxiliary modules.

$$\forall x\forall y[\text{sort}(x,y) \equiv \text{generated\_test}(x,y)] \quad (7)$$

$$\forall x\forall y[\text{permutation}(x,y) \equiv \text{generated}(x,y)] \quad (8)$$

$$\forall x[\text{sorted}(x) \equiv \text{tested}(x)] \quad (9)$$

By taking the program text of module generate_test out of the module library and replacing all occurrences of the auxiliary module name generated and tested with permutation and sorted, respectively, the user obtains the programs of the new module sort. It should be noted that this method does not verify the assumptions in Formulae (8) and (9). Therefore, modules having completely different functions may be retrieved by this method. However, as shown above, the programs of module generate_test can

easily be reused regardless of whether Formulae (8) and (9) are satisfied. Auxiliary modules permutation 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.

5.3 Conclusion

This section has presented a method for retrieving reusable software modules by verifying the equivalence of first-order predicate logical formulae given as their specifications. As long as the specifications can be stipulated by first-order predicate logical formulae, this method can be applied regardless of their 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 specifications 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 on 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 description of the specifications. Therefore, the objective of this method is not to verify the logical equivalence including specifications module (auxiliary predicates and data classes) being referenced, but to find similarities in their "reference forms." The method assumes that if the reference forms in specifications are similar, the interfaces based on the programs that implement them are also similar and, therefore, can be easily reused. In this method, the equivalence of specifications of auxiliary predicates is determined solely by how they are called and how parameters are given.

Using this method, a programmer can use any part of the reusable modules at each step of the stepwise refinement. The programmer can also further refine unusable parts.

6. Program Composition

6.1 Introduction

Program composition is based on reuse of modules which are retrieved from the module library.

6.2 Examples

Fig.3 shows an example of composing a program on_the_same_row by reusing on_the_same_column. The general composition process is as follows (the step inieces correspond to the ones in Fig.3):

(1) The programmer writes the specification of the module he wants using TELL/NSL.
(2) The system converts it to a set of logical formulae using the parser and invokes the functional retriever to find similar modules.
(3) If a module is found, its specification and program are presented to the programmer, with conditions for the module to satisfy given specification.
(4) The programmer determines whether the retrieved module is suitable.
If it is reusable, the module is registered in the module library after any required editing. The composition process is successfully completed.
(5) If no suitable module is retrieved, the programmer writes the program by himself, refines the specification dividing into some submodules, and repeats this process from (1).

(1) Given specification

```
Queen Q1 and queen Q2 are on the same column
 in 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 on the same column;
```

(2) Logical formulae of above

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

(3) Similar module retrieved

```
module:
 Queen Q1 and queen Q2 are on the same row
  in arrangement X means that
   1) The Y_coordinate of the position of Q1 in X
      is the Y_coordinate of the position of Q2
      in X.
 end on the same row;

logical formulae:
    ∀q1∀q2∀x[ on_the_same_row(q1,q2,x) ≡
         y_coordinate(position(q1,x)) =
         y_coordinate(position(q2,x))]

equivalent condition:
    ∀x[x_coordinate(x) ≡ y_coordinate(x)]

program of on_the_same_row: (in Prolog)
    on_the_same_row(Q1,Q2,X) :-
        position(Q1,X,P1),
        y_coordinate(P1,W),
        position(Q2,X,P2),
        y_coordinate(P2,W).
```

(4) Program of on_the_same_column composed

```
    on_the_same_column(Q1,Q2,X) :-
        position(Q1,X,P1),
        x_coordinate(P1,W),
        position(Q2,X,P2),
        x_coordinate(P2,W).
```

Fig.3 Example of program composition

Notice that the programmer must determine whether the retrieved module can satisfy his purpose; it is too difficult to make this decision automatically. The programmer must also modify the module's program correctly as required.

6.3 Conclusion

The functional retriever is so powerful that all the program composer has to do is interpret user's commands and control the flow of the process. Program composition is one application of the functional retrieval system.

The above process does not depend on the programming language. In the case of a logic programming language such as Prolog, it seems to be easier than for procedural programming languages to do step (4) automatically, but further research is required.

7. Explanation Generator

The explanation generator analyzes the programs of modules and generates natural language-like sentences that explain the logic of the programs. The explanation generator is to be used in design, creation, and maintenance of software, and makes it easier to understand the functions of modules to be reused.

The explanation generator currently uses rather simple methods. It has its own database that contains the templates of explanations of modules. A template consists of the name of the module, its parameters, and the skeleton of the sentence to be generated. The explanation generator initially has the skeletons for the built-in Prolog predicates, and users must register the templates for their predicates. TELL/NSL is used to make registration easy.

8. Conclusions and Future Work

Using Prolog, we have implemented a prototype software development consultation system. We have learned that it is feasible to use a well-defined subset of natural language as a software specification description language. A powerful method to retrieve software modules has also been presented.

We plan to develop a Japanese-language-like specification language, and evaluate it in applications.

ACKNOWLEDGEMENTS

This work is based on the results of the R & D activities of the fifth generation computer systems project. The authors would like to thank Dr. T. Yokoi of Institute for New Generation Computer Technology (ICOT) for his encouragement and support.

The authors also would like to thank Prof. H. Enomoto (currently Fujitsu International Institute for Advanced Study of Social Information Science), the members of Enomoto and Yonezaki lab. at the Tokyo Institute of Technology, and all the members of the TELL working group for their meaningful discussions.

REFERENCES

[1] M. Sugimoto, et al. "Design Concept for Software Development Consultation System," ICOT TR-071, 1984.

[2] H. Enomoto, et al. "NATURAL LANGUAGE BASED SOFTWARE DEVELOPMENT SYSTEM TELL," ICOT TR-067, 1984.

[3] H. Yoshida, et al., "RETRIEVAL OF SOFTWARE MODULE FUNCTIONS USING FIRST-ORDER PREDICATE LOGICAL FORMULAE," Proc. of LPC'85, Springer-Verlag, 1986. (to be published)

[4] Y. Matsumoto, et al. "BUP: A Bottom-Up Parser Embedded in Prolog," NEW GENERATION COMPUTING, vol.1, no.1, 1983.

[5] L. Pereira and D. Warren, "Definite Clause Grammar for Language Analysis -- A Survey of the Formalism and a Comparison with Augmented Transition Networks," Artificial Intelligence, no.13, 1980.

[6] C. Chang, et al. "Symbolic Logic and Mechanical Theorem Proving," Academic Press, 1973.