A CASE-BASED REASONING APPROACH TO PROGRAM SYNTHESIS

Yulia Korukhova and Nikolay Fastovets
Computational Mathematics and Cybernetics Faculty, Lomonosov Moscow State University
Leninskie Gory, GSP-1, Moscow, 119991, Russian Federation

Keywords: Automated Program Synthesis, Case-Based Reasoning, Ontologies.

Abstract: The paper deals with automated program synthesis. For program construction a case-based reasoning approach is used. The case library, organized as ontology, contains specifications and corresponding texts of already known programs. In the specification the relationship between inputs and outputs is written, the text of a corresponding program is written on a programming language. The specification of the desired program is taken as a task to find solution for, and we are looking for similar cases - specifications in the case library. If such a case is found we are trying to adapt the corresponding text of program. The main problems that occur in the implementation of the proposed approach are the following: the organization of case library, definition of similarity and ways of adaptation. We propose to keep the case library as ontology; the ALC is used to describe specifications. This representation helps to find similar specifications and to adapt the corresponding solutions.

1 INTRODUCTION

The problem called automation of programming was treated almost in the same time with the development of programming languages. First, the idea was to come to the higher level of abstraction: from programming in the machine codes and the direct use of assembler to the higher level of abstraction – to programming languages. The development of a program often starts from the specification – from a description what the program should do. Writing specifications instead of programs looks like the next level of abstraction. Good specification is are more clear and readable for humans. The development of automated synthesis systems aims to create such a language (or languages) for writing specifications that should be automatically transformed into programs.

There are several approaches to automated construction of programs. Deductive methods (Manna and Waldinger, 1992) intend the construction of programs simultaneously with proof of their specification. There are several synthesis methods, but the automation of deductive synthesis is a challenging task: the success depends not only on the correctness of the given specification, but also on the sufficiency of domain information (usually described as axioms or partially included in the synthesis rules) and on the order in which the derivation rules are applied.

The idea of this article was inspired by observations how students who started to learn programming solve some of the tasks. They have a course on basic algorithms; they know several examples of programs. If they are asked to solve a new task they sometimes they invent a new algorithm, but in many cases they try to adapt an example program they have to the requested one. The precondition for application of the last method is the following: the students should have a solution for some task that is considered to be similar.

In this work we are trying to model this process of solving tasks in programming. We are trying to implement case-based reasoning for construction of programs from their descriptions based on first-order logic.

The paper is organized as follows: in the section 2 the foundational methods and their application for a particular task of program synthesis and the problems that arises are discussed. In the section 3 our approach to the solution of the mentioned problems is presented together with our prototype system and some results are described. In the section

Korukhova Y. and Fastovets N..
A CASE-BASED REASONING APPROACH TO PROGRAM SYNTHESIS .
DOI: 10.5220/0003064903350338
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2010), pages 335-338
ISBN: 978-989-8425-29-4

4 the conclusions are drawn.

2 BASIC SYNTHESIS METHODS

In the task of synthesis we have a specification, that describes a relationship between inputs and outputs of a program (but not necessarily describes an algorithm for result computation) and we need to construct a text on some programming language, that implements a computational algorithm for an output that meets the conditions presented in the specification. We are going to explore a case-based reasoning approach.

2.1 Case-based Reasoning

The main idea of Case-Based Reasoning (CBR) is to model people model of reasoning and try to implement it on the computer system. Humans often use some set of known solutions for different tasks and they try to adapt these solutions for new problems they are faced with.

The general CBR is performed as a cycle of 4 steps (Aamodt, Plaza, 1994): RETRIEVE the most similar of the cases, REUSE the information and knowledge in the found case to solve the new problem (by adapting the solution of the similar case), REVISE the proposed solution, RETAIN the parts of this experience likely to be useful for future problem solving.

The main feature of this approach is the ability for self-learning system that allows to improve performance of work with case library and to improve adaptation methods. However, this approach also has some problems. First, the case library is very large on practice, and the significant delays because of search would make such a system useless. Secondly, the correctness of the system depends on the definition of similarity among precedents.

2.2 Ontologies and Description Logics

In our approach we propose organization of case library as ontology. So the mechanisms developed for ontologies can be used for comparing specifications of functions and help to find similar cases for the given tasks.

We can describe the ontology as a systematic structure of knowledge, which describes the classification of some objects and relationships between objects. In our case the specifications of already known functions are such objects.

The most important for us is the classification described in the so-called terminological box (TBox). Expressions (concepts) in such logics describe some subset of elements of the original set (the alphabet). We will consider one of such logics - ALC - which we use in our system.

2.3 Description Logic ALC

ALC is belongs to the family of description logics. In this section we recall syntax of ALC expressions (M. Bienvenu, 2008). Concepts expressions in ALC are built up from the set of atomic concepts C and roles R according to the following recursive definition

$$C ::= A \mid \neg C \mid C \wedge C \mid C \vee D \mid \forall R.C \mid \exists R.C \mid \top \mid \bot$$

where $A \in C$ and $R \in R$.

A concept C is said to be satisfiable if there is an interpretation (model) I such that $C^I \neq \emptyset$. If there is no such a model for the concept it is called unsatisfiable. A concept C is said subsumed by D (or D subsumes C) if for every model I $C^I \subseteq D^I$. A concepts C and D equivalent if C subsumes D and D subsumes C. The ALC is used for internal description of specifications in our system.

2.4 Using of Ontologies in CBR

The organization of a knowledge base as an ontology is very useful for storing data and also in CBR. The system Taaable (F. Badra and others, 2009) can be considered as an example. In this system an ontology is used to store information on a set of known recipes, as well as for selection of possible ways of adaptation of known solutions to new problems.

The hierarchy concept used in the ontology allows to reduce the number of possible precedents. A description of the concept itself provides an opportunity to advance the conclusion of the existing differences between the well-known precedents and purpose. Thus, this approach helps to organize our case library hierarchically that helps both in the search and in the adaptation process.

We are going to use the similar idea for organizing the information about programs and their specifications.

3 IMPLEMENTATION AND CURRENT RESULTS

In this section we observe implementation of CBR, where case library is organized as ontology, for solving a task of program synthesis and its practical realization in our prototype system.

3.1 Knowledge Base Organization

Main part of our knowledge base is hierarchy of concepts, each of them corresponds to some relation (predicate or specification). The functions are associated with some concept via a corresponding predicates. Also, the system uses library of known programs and every specification description if our hierarchy associated with one or more programs in this library. We use the terminology boxes of the ontology to perform search and to compare specifications. Hierarchical organization of the ontology allows to reduce the search time, which is an important advantage for CBR approach.

3.2 Common Model of Work

The general CBR cycle is represented in our system in the following way. The work starts with a first order logic specification of some program. Then the search of the most similar specification (of already known program) is performed. At the next stage the adaptation of the solution – program is performed. The adapted program is given to user as a result of work. After user's estimation (whether the solution was good or not) the appropriate information is added to system libraries.

3.3 Building of a Function Concept

After receiving a specification from user, the system should search for similar ones among the known functions. Thus, our first task is to build a sufficiently informative description from the specification function.

Our translation procedure consists of several steps. First of all, we build disjunctive normal form of the specification. Secondly, in every disjunction we perform elimination of function calls. We replace function calls for each clause D in the obtained expression S' by new variables (that don't occur free in the expression before this stage) and add predicates that corresponds to these functions. After that we construct an appropriate ALC concept for the function. In such description we will use special roles: *Contain*, *ContainNeg*, *Variant* and the family of roles $Param_1$, …, $Param_n$. Also, we will use special concepts, which describe the roles of variables in the specification: *Output*, $Input_1$, …, $Input_m$, $Variable_1$, …, $Variable_n$. First of all we associate every variable in S' with concept of form (*VarRole* /\ *VarType*), where VarRole is one of special concepts Output, Input1 etc. and VarType is concept of domain for variable. Then, for each atom $A(x_1, …, x_k)$ we build a description

$AC = cA /\ \exists\ Param_1.x_1 /\ … /\ \exists\ Param_L.x_L$,

where cA is concept associated with predicate A and variables x_1, …, x_k replaced with variable descriptions (as above).

Next step is building of concept for conjunction. We describe conjunction as set of intersections between concepts P_1, …, P_L, each of them has form ∃ Contain.AC, if corresponding atom contain in expression with positive polarity, or ∃ ContainNeg.AC – if it has negative polarity. So, we get conjunction description

$CC = \exists\ Contain.AC /\ … /\ \exists\ ContainNeg.AC' /\ …$

If the expression contains a disjunction then we build descriptions for each disjunct separately and include them in the common description:

$\exists\ Variant.CC_1 /\ \exists\ Variant.CC_n$.

Thus we obtain a complete description of specification S. This description is used to compare with other specifications from the case library.

3.4 Difference Concept Construction

To store and use our knowledge about adaptation for programs we use another kind of descriptions - the differences concepts, which are also written on ALC and organized as a hierarchy. Every such concept describes a subset of adaptation rules which eliminates the corresponding differences in specifications and transforms the program text from the case library. We build such differences descriptions as concepts with special roles: Add and Remove or their intersection. Let we have two concepts: A = A1 /\ … /\ An and B = B1 /\ … /\ Bm.

We construct differences description D in the following way: $D = Add.B_{i1} /\ Add.B_{ik} /\ Remove.A_{j1} /\ Remove.A_{jp}$ where $\forall\ t,\ 1 \le t \le k\ \nexists\ g,\ 1 \le g \le n$ such that $B_{it} = A_g$ and $\forall\ t,\ 1 \le t \le p\ \nexists\ g,\ 1 \le g \le m$ such that $A_{jt} = B_g$.

So we describe addition of subconcepts from B, which doesn't occur in A, and elimination of subconcepts from A, which are not in B.

3.5 Adaptation

For the similar (but not the same) cases we usually need to build a concept of differences and to adapt the solution. Our adaptation mechanism uses hierarchy of differences concepts and a list of

adaptation rules, each of them is associated with one of differences concept.

During the procedure of adaptation our system performs search of most similar differences description for specified one. If such description is found, the system applies the rewriting rules associated with it to the program corresponding to the most similar specification. In the other case, system tries to split differences description via intersection and performs search for every conjunct.

3.6 Concepts Comparison

Our comparison algorithm based on structural subsumption (F. Baader, W. Nutt, 2003). We use next comparing rule: for two concepts $A = A_1 /\backslash \ldots /\backslash A_n$ and $B = B_1 /\backslash \ldots /\backslash B_m$, if $\forall$ i, $1 \le i \le n$ ($\exists$ j, $1 \le j \le m$ such that $A_i \subseteq B_i$) then $A \subseteq B$. So, our subsumption checking algorithm based on this rule return answer $A \subseteq B$ iff for each subconcept from A there is a subconcept in B that subsumes it. Respectively to this assessment we build our functions and differences concept hierarchy. So, we can postulate that every concept C that subsumes by some concept D contain all restrictions from D. Also, we define similarity between concepts C and D as the length of path in the hierarchy tree from C to D. According to this the most similar concept for a given one (functions or differences) is it's immediate predecessor in our library.

3.7 Search Algorithm

Our search algorithm uses the defined above subsumption assessment as an heuristic in breadth-first search. The target concept and the root of hierarchy tree (or another node, which can be used as root of subtree) are inputs in the process. At the every step we check matching between the root concept and the target. If our search procedure finds exact matching, it returns the root. In the other case, it checks the subsumption assessment between target and all the direct descendants of the root. If one of the descendants subsumes the target, our search procedure performs recursive call. If there are no target subsumers between root descendants, procedure returns root as the answer (as the most similar concept).

4 CONCLUSIONS

In this paper we consider one of the possible applications of Case-Based Reasoning in the program synthesis problem. Using of ontologies here allows us to organize search in the case library in a reasonable time and helps to define cases' similarity. The same idea applied to differences concepts helps to formalize the search of differences and perform adaptation. The ideas were tested on several examples, but the work is still in progress. First, we are going to extend the case library. Another point of improvement is the way of verification of the obtained program. Now this question is solved by user, but it is planned to be also automated. We are still far away from fully automated synthesis, but this approach makes a step to this goal

ACKNOWLEDGEMENTS

The work has been partially supported by RFBR grant 08-01-00627.

REFERENCES

A. Aamodt, E. Plaza, 1994. Case-Based Reasoning: Foundational Issues, *Methodological Variations, and System Approaches*, AI Communications. IOS Press, Vol. 7: 1

M. Bienvenu, 2008. Prime Implicate Normal Form for ALC Concepts. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*.

F. Baader, W. Nutt, 2003. Basics Description Logics. *The Description Logic Handbook: Theory, Implementation and Application*.

F. Badra, J. Cojan, A. Cordier, J. Lieber, T. Meilender, A. Mille, P. Molli, E. Nauer, A. Napoli, H. Skaf-Molli, Y. Toussaint, 2009. *Knowledge Acquisition and Discovery for the Textual Case-Based Cooking system WIKITAAABLE*. 8th International Conference on Case-Based Reasoning - ICCBR 2009, Workshop Proceedings, Seattle : United States

Manna Z. and Waldinger R., 1992 Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8): 674-704