# AUTOMATIC PROGRAM GENERATION FROM SPECIFICATIONS

# USING PROLOG

ALEX PELIN
School of Computer Science
Florida International University
Miami, Florida 33199

PAUL MORROW
AFWL / SCR
Kirtland AFB, NM 87117

## Abstract

This paper describes an automatic program generator which creates PROLOG programs from input/output specifications. The generator takes as input descriptions of the input and output data types, a set of tests, a set of transformations and the input/output relation. Abstract data types are used as models for data. They are defined as sets of terms satisfying a system of equations. The tests, the transformations and the input/output relation are also specified by equations.

The program generator creates a PROLOG program which takes as input a data item, item1, of the input data type and outputs a data item, item2, of the output data type such that the input/output relation is satisfied by the pair item1,item2. In building the program the generator uses only the tests and the transformations given as input. The program generator was writen in PROLOG. It was able to generate correct PROLOG programs for sorting lists with and without eliminating duplicate elements. The system contains specifications of the abstract data types natural number, boolean, list and array. The system can be used in two modes: in the first mode the user defines his/her own data types and in the second mode he/she uses the definitions that are already in the system library. A user interface is being constructed for the second class of users.

The paper also presents several methods for validating the input/output specifications. Some of them were implemented in the system. Descriptions of the heuristics employed by the program synthesizer are also included. Finally, the paper compares this system with the approaches taken by other researchers in automatic program generation : Prywes, Manna and Dershowitz.

## Introduction

This paper describes a project whose goal is to create a knowledge base for automatic program generation from input/output specifications . By its scope the project falls into the category of program synthesis. The goal of program synthesis is two-fold : to develop specification languages in which one can describe programming tasks and to develop heuristics which translate these specifications into high level language code.

The system described in this paper took as input the following items:

1. A description of the input data type;
2. A description of the output data type;
3. A description of the input/output relation;
4. A set of transformations;

5. A set of tests.

Figure 1

Based on the input data, the system tries to generate a program P which takes as input an item i of the input data type and outputs an item o of the output data type such that the input/output relation holds for the pair i,o. The program P is constructed by using only the tests and the transformations given as input to the program synthesizer. For example, the problem of sorting a list of natural numbers in increasing order is specified by the 5 items shown below.

1. The input data type is list of natural numbers;
2. The output data type is list of natural numbers sorted in increasing order;
3. The input/output relation is standard;
4. The set of transformations is inversion of consecutive elements;
5. The set of tests consists of the predicate SORTED, which checks if a list of natural numbers is sorted in increasing order, and the predicate less which checks if two consecutive elements in the list are in the right order,i.e. the first element is less than or equal to the second element.

Figure 2

Two data items i and o satisfy a standard input/output relation if o is obtained from i by applying a sequence of transformations. Each transformation in the sequence must be a member of the set listed in item 4 of figure 1. The tests listed in item 5 can be used to guide the transformations. The program synthesizer was able to generate a correct PROLOG program for the specifications shown in figure 2.

The paper is organized as follows: issues pertaining to the problem of validating program specifications are presented in section 2, heuristics used to generate PROLOG code are described in section 3 and the user interface is presented in section 4.

## 2. Validation of Specifications

The system uses abstract data types ([3]) as models for data. A data type is a set of terms, freely generated by some operations, which satisfies a set of equations (conditional equations). For example, the data type list of natural numbers consists of the data types :natural number, list of natural number and boolean. The terms of the data type are generated by the productions (rules) shown in figure 3.

```
natural ---> 0
natural ---> successor(natural)
list    ---> []
list    ---> [ natural | list]
boolean ---> True
boolean ---> False
boolean ---> natural <= natural
```

Figure 3

The variable 'natural' generates all the natural numbers, the variable 'list' generates all lists of natural numbers and the variable boolean produces all terms of type boolean. The operator 'successor' is the successor operator on the set of natural numbers. The semantics of the data type is given by the set of (conditional) equations presented below.

1. n <= n = True
2. n <= m = True ---> successor(m) <= n = False
3. n <= m = True ---> n <= successor (m) = True

Figure 4

The equations given in figure 4 describe the relation <= on the set of natural numbers. Since the user specifies the data types that are used by the system, it is fundamental that the system validates these specifications. An example of a validation problem is the following : for all natural numbers m and n, either m <= n = True or n <= m = False. In this system, data type validation questions become theorem proving problems. A data type validates a property if that property is a theorem produced by the system of axioms that define the data type. There are several methods that can be used to accomplish this goal. One can use first order logic, term rewriting systems and induction. The system described in this paper uses term rewriting systems as the main tool for validating data types. The system has a Knuth-Bendix completion procedure ([7]) that can be used to generate a complete set of reductions for a system of equations. At present the completion procedure operates only with pure equations but there are ways of extending it to conditional equations([5]). Several experiments were made with the interactive theorem prover ITP. There were many problems with the use of ITP for validating data type specifications. This theorem prover, like all theorem provers based on resolution, tends to generate a tremendous amount of useless clauses. It is therefore imperative to develop heuristics that eliminate clauses that are irrelevant to the proof of the question to be validated. The other problem is that the relation between first order logic and PROLOG is a complicated one. The translation of a set of first order logic sentences into a set of Prolog clauses is not easy. The third problem with using first order logic to validate questions in this system is that, since the data types are inductively defined, theorems that require induction cannot be proved. For example the theorem which states that for all natural numbers m, m <= successor(m) requires induction. Since the progress in automating induction has been slow, the system does not use direct inductive methods.

## 3. Heuristics Used by the System

So far the system can deal with problems in which the output data type is a subset of the input data type. Sorting problems fall in this category. The data type list of natural numbers sorted in increasing order can be defined as the subset of the data type list which satisfies the predicate SORTED described below:

1. SORTED([])
2. SORTED([m])
3. m <= n = True ---> SORTED([m |[n |list]]) = SORTED([n |list])
4. m <= n = False --->-SORTED([m|[n|list]])

Figure 5

In figure 5 - stands for negation. The condition m <= n = True, m <= n = False correspond, respectively, to tests -less(n,m) and less(n,m) shown on line 5 of figure 2.For the class of problems in which the output data type is a subset of the input data type and the input/output relation is standard, the system focuses on the predicate that defines the output data type. For the problem described in figure 2, the system transforms the third clause of the specification shown in figure 5 into two clauses:

```
5. m <= n = True , SORTED([n|list]) ---> SORTED([m|[n|list]])
6. m <= n = True ,-SORTED([n|list]) --->-SORTED([m|[n|list]])
```

Figure 6

Then the system employs a method which focuses on the negative clauses from figures 5 and 6. If SORTED(list) is false then the clause -SORTED(list) must occur either at the right of the ---> sign in clause 4 of figure 5 or at the right of the ---> sign in clause 6 of figure 6. The system assigns higher priority to clause 6 than to clause 4. In general clauses that contain recursive calls have higher priority than those that do not have them. If -SORTED(list) is derived from rule 4 of figure 5 then the system establishes the negation of the condition m <= n = False as its goal. In this case the system looks for a transformation, or a sequence of transformations , T_list such that: if -SORTED(list) is obtained from rule 4 then the precondition of rule 4 does not apply to T_list(list). In this particular case the system looks for a transformation that carries the argument [m|[n|list]] of the rule 4 of figure 5 into a list which is sorted, or it has length less than the original argument, or it is of the form [p|[q|list']] and p <= q = False does not hold. The system has a generate and test algorithm for finding T_list. The states are pairs of the type <Term, Tramsformation>, where Tranformation is a sequence of transformations that brings the original argument to the list Term. It uses various guiding functions for searching the state space. The method was tested using the sets of transformations {EXCHANGE} and {EXCHANGE,REDUCE}.The equations for EXCHANGE and REDUCE can be found in figure 7.

```
1. EXCHANGE([m|[n|list]] = [n |[m |list]]
2. REDUCE([m|[m|list]) = [m |list]
```

Figure 7

The system produced correct answers in both cases. The predicates must be defined in a hierarchical manner. This means that test1 can be defined as a function of test2, but test2 cannot also be defined as a function of test1.

## 4. The User Interface

The system can be used in two ways. In the first mode the user defines his own data types, tests, transformations and input/output relation. In the second mode the user employs the definitions that are already in the system library. The system has an interface that allows the user to enter commands in English. The user can enter statements like the ones shown in figure 8.

```
1. X is a real array
```

2. 3 is the size of X
3. Y is X reversed
4. Display Y

Figure 8

These types of instructions allow users to employ the system like a calculator. Work is under way to enrich the interface to the point where it can accept atatements like the ones shown in figure 2 . In this case the user will receive from the system the program that accomplishes the task described in English.

## 5. Conclusions

Automatic program generation is an important problem in automating the software development cycle ([1],[4]). It can be used do develop program modules from task specifications. Defining specifications languages is a difficult job ([6],[10]). Validating the specifications is more difficult. The system uses equations for specifications. The system MODEL, developed by the group led by N. Prywes ([9]), also uses equations for specifications. Dershowitz ([2]) employs term writing systems to synthesize programs. In the system presented in this paper, term rewriting systems are used for validating specifications and as an intermediate step in translating equations into PROLOG clauses. Dershowitz uses Pascal to carry out program synthesis. PROLOG seems better suited for implementing heuristics. The programs generated by the system are quite different than those which are constructed through informal means. This fact is mentioned by Manna ([8]).

## Acknowledgement

This work was supported by the Air Force Office of Scientific Research/AFSC under contract F496-C-0013.

## Bibliography

1. Boehm, B. : 'Improving Software Productivity', Computer, Vol. 20, No. 9, September, 1987, pp. 43-57.
2. Dershowitz, N. : 'Synthesis by Completion', proceedings of IJCAI-85, Morgan Kaufmann, 1985, pp. 208-214.
3. Ehrig, H. and Mahr, B. : Fundamentals of Algebraic Specification 1, Springer Verlag, 1985.
4. Frenkel, K. :'Towards Automating the Software-Development Cycle', CACM, Vol. 28, No. 6, June, 1985, pp. 578-591.
5. Ganzinger, H. : 'A Completion Procedure for Conditional Equations', University of Dortmund Technical Report 234, October,1987.
6. Hoare, C. :'An Overview of Some Formal Methods for Program Design',Computer, Vol. 20, No. 9, September, 1987, pp. 85-91.
7. Knuth, D. and Bendix, P. : 'Simple Word Problems in Universal Algebras', Computational Problems in Abstract Algebra, Pergamon Press, 1970, pp.80-149.
8. Manna, Z. and Waldinger, R. :'The Origin of the Binary Search Paradigm', proceedings of IJCAI-85, Morgan Kaufmann, 1985, pp. 222-224.
9. Prywes, N. , Shi, Y. ,Szymansky, B. and Tseng, T. :'Supersystem Programming with Model' , Computer, Vol. 19, No. 2, February, 1986, pp. 50-60.
10. Roman, G. : 'A Taxonomy of Current Issues in Requirements Engineering', Computer, Vol. 18, No. 4, April, 1985, pp. 14-22.