Domain-Based Program Synthesis Using Planning and Derivational Analogy

Sanjay Bhansali

This research was motivated by the widely held belief that constructing an automatic program synthesis system can accept a high-level description of a problem for an arbitrary domain and generate code for the problem completely automatically is pragmatically impossible. However, by focusing on a well-defined domain, it is possible to incorporate sufficient knowledge within a system so that it can communicate with an end user at the level of his/her application and automatically generate a program from a problem specification. Such knowledge-based systems often employ a catalog of transformational rules that progressively refine an abstract specification into a concrete implementation. A major research issue in such systems is how to increase the efficiency of the systems by controlling the application of rules and avoiding repetitive traversal of the search space.

In my Ph.D. dissertation (Bhansali 1991), I develop an integrated knowledge-based framework for efficiently synthesizing programs by bringing together ideas from the fields of software engineering (software reuse, domain modeling) and AI (hierarchical planning, analogical reasoning). Based on this framework, I constructed a prototype system, APU, that can synthesize UNIX shell scripts from a high-level specification of problems typically encountered by novice shell programmers. An empirical evaluation of the system’s performance points to certain criteria that determine the feasibility of the derivational analogy approach in the automatic programming domain when the cost of detecting analogies and recovering from wrong analogs is considered (Bhansali 1991; Bhansali and Harandi 1991; Harandi and Bhansali 1989).

System Overview
Chapter 2 of the dissertation gives an overview of the framework and describes its two major components—the knowledge base and the program generator. The knowledge base consists of three subcomponents: a concept dictionary, a library of reusable components, and a layered rule base. The concept dictionary contains a description of the domain-dependent vocabulary (objects, functions, and predicates) needed for describing problems at a high level. Using this vocabulary, a user can communicate with the system at the level of his/her application. The concepts are organized in an abstraction hierarchy with inheritance of properties from higher levels to lower ones. Besides providing an economic representation, this hierarchy also forms the basis for the system to recognize analogies among problems.

The library of reusable software components is used by the program generator to avoid the repetitive synthesis of programs or program fragments that are used frequently. Two types of software components—subroutines and templates—are static components of the library. A third component—derivation history—is acquired dynamically by the system as it solves problems specified by the user. A derivation history is a trace of how a program is derived from a specification and is used to speed the synthesis of analogous programs.

The layered rule base contains a catalog of strategy rules, problem-solving rules, and domain-specific rules that can be employed to transform a problem specification into a shell script. The layered structure of the rule base should allow the system to be configured to different domains and target languages with minimal changes.

The program generator consists of two subcomponents: a planner and an analogical reasoner. The planner, based on the concept of hierarchical planning, uses the layered structure of the rule base and certain heuristics to ensure that plan failures are detected early and that efficient plans are chosen over inefficient ones. The planner uses a top-down decomposition approach to first generate a plan to solve the problem (in which the various UNIX commands and subroutines form the primitive operators) and then uses a set of simple transformations to convert the plan into a shell program.

The analogical reasoner, based on the derivational analogy paradigm of Carbonell (1983), complements the role of the planner. The analogical reasoner can automatically retrieve problems from the derivation history library that are analogous to the current problem; replay the derivation history to synthesize programs more efficiently than the planner; and, finally, store the derivation traces of solved problems in the library to be used in a future context.

Chapter 3 describes the planner-based program synthesis algorithm of APU. The algorithm is illustrated through an example, showing the various kinds of general and domain-specific knowledge needed.

Analogue Reasoner

Although the idea of an integrated framework is advocated in the thesis and implemented in the prototype system, I have been most concerned with demonstrating the effectiveness of the analogical reasoning capability of the system. The three crucial processes in an analogical reasoning system that replays previous derivations are (1) retrieving an appropriate analog from a case library of derivation histories, (2) adapting the derivation to fulfill the requirements of the new problem, and (3) consolidating the result back in the case library. In my dissertation, I describe and implement the first two processes of retrieval and adaptation and provide suggestions on how the result might be consolidated into a case library.
Chapter 4 describes the methodology for recognizing and retrieving analogs. The recognition algorithm is based on a set of four heuristics that consider various abstract features of problems in order to detect analogies. The features considered are designed to estimate the top-level solution strategies, as well as the sequence of individual problem-solving steps, by analyzing the outermost constructs of problem specifications, the relationship between input and output arguments in terms of the systematicity principle of Gentner (1983), various syntactic cues in the problem specification, and the degree of similarity between corresponding objects in a target problem and a candidate analog. The organization of concepts in the concept dictionary is crucial to effecting the operation of the heuristics in determining the best analogous program for a target problem. The heuristics themselves are dependent, to a large extent, on the representation of the problems as well as the domain, but the basic principles on which they are based seem to be applicable to several domains and representations.

Chapter 5 describes the replay algorithm and illustrates it through an example. Several theoretical issues pertinent to replay systems have been detailed by Mostow (1989). I discuss some of the relevant issues in the context of APU, comparing my approach to some of the other works in the field.

Empirical Evaluation
Chapter 6 contains a detailed account of experiments conducted to perform an empirical evaluation of the system's performance. The experiments were designed to assess APU's performance by determining the answers to the following questions: Is it feasible to detect analogous problems using domain knowledge? Does the cost of detecting analogies and recovering from an inappropriate analog outweigh the benefits of adapting an existing program (as opposed to synthesizing a program outright)? How is the performance of the system affected by the size of the derivation history library?

The data set for the experiment was collected by randomly selecting problems from a subdomain of UNIX using fixed procedures. APU's performance was measured by using it to synthesize shell scripts for the problems with various combinations of the retrieval heuristics and without analogy. The results of the experiments show that when using all 4 retrieval heuristics, APU's retrieval capability is almost as good as a human's (APU missed the best analog in only 2 out of 45 cases). The time to synthesize programs using analogy was found, on average, to be about half the time it takes to synthesize programs without analogy. Note that this performance is the average case; in some cases, the use of analogy actually degraded the system's performance. A promising observation was that when APU found an analog, the reduction in time was much greater than the increase in time when APU found no appropriate analog. Finally, the time taken to search for analogous problems depends roughly on the number of features used to index problems to the number of problems in the library, which suggests when it would be appropriate to store problems in the derivation history library.

In chapter 7, I discuss some of the other replay-reuse systems: POPART (Wile 1983), BOGART (Mostow, Barley, and Weinrich 1989), XANA (Mostow and Fisher 1989), KIDS (Goldberg 1990), DMS (Baxter 1990), PRIAR (Khammepati 1989), PRODIGY (Carbonell and Veloso 1988), and others. Among these, POPART, XANA, KIDS, and DMS are automatic programming systems. However, the emphasis in all of them is on design iteration rather than analogical mapping; hence, the issue of analogical retrieval is side-stepped.

Conclusions
The belief that reuse is the central mechanism that will foster large-scale improvement in software-engineering productivity is widespread. This thesis represents a continuation of the efforts of several researchers engaged in making reuse a practical reality. The thesis adopts a wide-spectrum view of reuse, in which domain-

The time to synthesize programs using analogy was... about half the time it takes to synthesize programs without analogy.

Acknowledgments
This work benefited greatly from the guidance of Mehdi T. Harandi and Uday Reddy and the suggestions and criticisms of Jack Mostow, Gregg Collins, and the members of the Knowledge Based Programming Assistant (KBPA) Group at the University of Illinois at Urbana.

References


Sanjay Bhansali is a research associate at the Knowledge Systems Laboratory, Stanford University. He obtained his Ph.D. at the University of Illinois at Urbana-Champaign. His research interests include knowledge-based software engineering, knowledge acquisition, redesign, analogical reasoning, and machine learning.