Automated Code Generation Using Case-Based Reasoning, Routine Design and Template-Based Programming

Yuri Danilchenko and Richard Fox

Department of Computer Science
Northern Kentucky University
Nunn Drive
Highland Heights, KY 41099
{danilcheny1, foxr}@nku.edu

Abstract

Automated code generation is the process whereby a computer program takes user specifications in some form and produces a program as output. Automated code generation can be the process undertaken by a compiler, which generates an executable program from a source program, but it also applies to the situation where the input is a task described at some level of abstraction and the output is a program that can perform that task. Several different approaches have been utilized to varying degrees of success to automate code generation, including Case-Based Reasoning, formal methods and evolutionary algorithms. In this paper, a system is introduced which combines Case-Based Reasoning, Routine Design and Template-Based Programming to generate programs that handle straight-forward database operations. This paper presents the approach taken and offers some brief examples.

Automated code generation (ACG) is the process whereby a computer program takes user specifications in some form and produces a working program as output. When the user input is some abstract description of a task, as opposed to source code in some high level language, ACG presents both a challenging problem and an opportunity to reduce cost. Automating the programming task, which normally requires a great deal of expertise involves employing techniques that comprise design or planning, logic and programming knowledge. The benefits of ACG include reducing or eliminating expenses involved in software development and maintenance, which studies have indicated could cost corporations as much as 10% of their yearly expenses (Jones 2010).

One form of ACG is the compiler. The user provides source code as input and the compiler generates an executable program. Compilers have been in regular use since the late 1950s when the first high level languages were developed. Although initially many programmers scoffed at the idea that ACG could produce efficient and correct code, very few programmers today would write code in a low level language, favoring the high level languages and compiler technologies. However, the compiler requires too detailed an input as the programmer must still produce the algorithm in a proper syntactic form.

In Artificial Intelligence (AI), a variety of approaches have been explored to support software development. Case-Based Reasoning (CBR), for instance, can be used to maintain a library of code routines (e.g., objects, methods), select the code routines that best match user specifications, and present the those options to the software developer. Alternatively, through genetic and evolutionary programming, code can be mutated and tested for improvements. If improved, the new code becomes a base for the next generation of code. Random changes can potentially lead to code that is more concise, more efficient, or more correct.

At present, neither CBR nor evolutionary approaches has yielded an ACG system that can replace a software developer. In this paper, the research focuses on three different but related areas. First, code generation is thought to be a design problem. A solution will be a plan. Plan steps can be specified at a generic level and then refined into more detail. Eventually all plan steps will be filled in with code components from a component library. Once selected, these components are combined and used to fill in a program template.

Although the word “plan” is being used, the plan is a description of a solution, or a design to solve the stated problem. The plan steps provide goals to be fulfilled. Code components are selected to fulfill each of these goals. Thus, the problem is one of designing a solution through a code component library.

The plan itself is retrieved from a library of plans, based on user specification. Additionally, if the selected plan does not precisely match the user specifications, alterations can be made. The refined plan can be stored for future retrieval. Thus, an ACG system can be built as a combination of Routine Design (RD), Template-Based Programming (TBP) and Case-Based Reasoning (CBR).

In this paper, the Automated Coder using Artificial Intelligence (ACAI) system is presented. The paper is laid out as follows. Section 2 identifies related work and some background into both RD and CBR. Section 3 presents the ACAI system. Section 4 contains a brief example and description of the system in action. Section 5 offers some conclusions and future work.

Related Work

The earliest instance of CBR is found in the system CHEF (Hammond 1986), a program to generate Szechwan cuisine recipes based on user goals. CHEF utilized a library of previous dishes for cases. Cases included such pieces of information as ingredients, textures, and preparation instructions. The CHEF system would retrieve a closely matching recipe based on the user specification, compare the goals of the matched recipe to the user's specifications, identify goals that were not met, or constraints that would not be met, and attempt to repair the selected recipe. The new recipe would be stored in the library so that the system could *learn* over time. This initial CBR system demonstrated the utility of the approach: solving new problems through previous solutions. A CBR system would perform four primary tasks: case retrieval, case reuse, case revision, caseretention.

The Kritik system (Goel, Bhatta and Stroulia 1997) developed in the late 1980s applied CBR to the physical design problem. Cases would represent component parts and Kritik would propose a design for a physical artifact by selecting components. Unlike CHEF where cases were represented by goals, Kritik represented its cases by their structure, function and behavior. The components' structures would be used to ensure that the components did not violate constraints, components' functions would be used to match goals, and components' behaviors could be used in simulation to ensure that the device functioned as expected. CHEF and Kritik are noted for their contribution to CBR although neither addressed ACG.

The Individual Code Reuse Tool, or ICRT, applies CBR to software reuse (Hsieh and Tempero 2006). A library of software components comprises the cases for the system. In ICRT, the software components are represented by both complete code segments and incomplete or partial code segments, the latter of which may be syntactically invalid as is. Cases are stored in a flat structure and indexed using attribute-value pairs. Indexes are assigned by the software developers using the system. Components are selected using a nearest-neighbor algorithm and brought to the developer's attention. It is up to the developer to utilize the suggested code segment or not. Therefore, while CBR is used, it is not an automated system. Of particular note however is the indexing scheme. Case attributes are represented using functionality cards, describing for each code segment the segment's language, feature, property and description.

In the Software Architecture Materialization Explorer (SAME) system, the goal is to produce object-oriented designs (Vazquez, Pace and Campo 2008). These designs are then presented to the developers who use the designs to produce the final programs. The designs are produced from a case library of various software architectural parts, such as a data access layer. Although the developers modify the case components by hand, SAME monitors any such operations to capture the changes for future uses.

The Case-Based Reasoner for Software Component Selection (Fahmi and Choi 2009) is currently only a conceptual design of a CBR system for software component selection. As with the previous two systems, this system automates only the selection of case components from a library of reusable software components. Cases include function, associated components, component justification and case justification in support of providing rationale for why a component might be used.

While the previous systems automated only a portion of the process, the Case-Based Software Reuse System, or CAESAR, (Fouqut and Matwin 1993) offers an example of a complete ACG. CAESAR applies a variant of CBR called compositional software reuse to perform code generation in the domain of linear algebra. Cases are reusable mathematical routines written in C. Code segments are retrieved based on user specifications and partial matching, along with plan decomposition. Inductive logic is used to capture frequently occurring instances of code segments so that these can be stored for future use. Such groupings are called slices.

Finally, Menu Browser Using Case Based Reasoning (MESCA), applies CBR to the problem of generating a user interface based on reusable software components (Joshi and McMillan 1996). Here, the reusable components are menus and the system will adapt menus to fit specified functions, application types, user-tailored fields and graphical design.

Aside from a great number of CBR efforts, the ACAI system highlighted in this paper draws from both RD and TBP. RD (Chandrasekaran and Josephson 2000) is a class of design problem in which the overall design strategy is well known and can be represented through plan decomposition. That is, solving an instance of the design problem is handled by decomposing the problem into subproblems or components. Each component itself might be further decomposed.

In RD, at the lowest level, specific design steps are available as component descriptions. A component description defines in English, through code, or mathematically how a given component is constructed and placed into the overall design. Commonly, there are multiple component descriptions available for any component. Therefore, the best component description is selected using some form of matching knowledge based on user specifications, constraining factors, and demands imposed by other components. RD has been applied to numerous problems from physical design (air cylinders) to abstract planning (air force mission planning) and abstract design (nutritional meal design) (Brown and Chandrasekaran 1989, Brown 1996, Fox and Cox 2000).

Template-based programming (TBP) originated in the 1960s but came into use primarily in the 1990s. The idea is to represent program logic in a generic form that can be filled in later by another program. For instance, a loop might be represented generically only to have its details filled in at a later time when those details become known. TBP has been applied to a number of problems ranging from the numeric subroutines to web site generation (Fernandez et al 1993, Jiang and Dong 2008).

An Automated Coding System: ACAI

The Automated Coder using Artificial Intelligence (ACAI) system is a first pass at a purely automated code generation system (Danilchenko 2011). Code generation systems cited in the previous section either required human involvement in the processing loop or were restricted to domains that may not be amenable to a general case, such as creation of menus and linear algebra. It is envisioned that the approach taken by ACAI can extend to a great number of applications and domains, although currently ACAI only solves database-type problems. Specifically, the initial implementation of ACAI was constructed to tackle the queries listed below. These queries were identified by data analyst at a hospital, citing that software which could solve such tasks would greatly reduce their workload.

- Average, maximum, minimum patient length of stay, by diagnosis, age, department
- Average amount of time patients waited between arrival and first procedure, first lab test, first physician visit, first triage
- Search for all patients who meet a given mode of arrival (ambulance, car, walk-in, air-transport) sorted by arrival time
- Average, maximum, minimum time to get lab results over all patients and lab requests
- Average, total, mean number of patients with/without insurance by day, week, month, year
- Most common diagnoses by time of day, weekday, month or season
- Number of patients by doctor, unit, nurse, diagnosis, location, age
- Average, mean amount of time between preliminary finding and final lab result

The restriction to the medical domain was made because of the interest in the topic. The limitation to handling database-type operations was made to ensure that a prototype system could be constructed. See section 5 for comments on future work.

ACAI accepts two forms of user input, the goal (i.e., the query or queries to be answered) and specifications for how to achieve the goal (e.g., computational complexity, memory and disk usage, form of input, form of output). The output of ACAI is a working Java program.

Given user input, the first step that ACAI undertakes is similar to that of CBR. A case must be retrieved from the library of cases. In ACAI, cases are *plans*, described using XML.

ACAI selects a plan through simple matching of user's stated goal for the program. ACAI contains plans for such activities as sorting, filtering, computation, and reasoning over event durations. As each plan is generic in nature, the queries listed above can be solved by just a few plans. Even so, the user's goals may match multiple plans, in which case ACAI uses a combination of matching plans rather than selecting a single plan.

A plan comprises several sections. First, the plan has a name and a description. Next, a plan has a number of steps broken down in three distinct types: input, operation, and output. Input steps describe from where the program will obtain its input. Operation steps describe the individual, executable portions that must make up the program to solve the given problem. Operation steps include a variety of types of computations such as summation, average, or maximum. Finally, output steps describe where the program will send its output. Notice that input and output steps describe the "where" while the operation steps describe the "how". Each step of a plan is described in terms of goals to be fulfilled. The goals are a list of attributes that describe the code that should be used to implement the given plan step.

Figure 1 provides an example of the input portion of a plan. This section contains two types of inputs. First are the generation inputs. This input allows ACAI to query the user who is generating a program, not the end user. Such input might, for instance, obtain information about the functionality of the intended program. For example, the user might input a specific type of aggregate function such as average or maximum. This input helps specialize a plan step, for instance altering the goal [Utilities – Aggregate – Property] into [Utilities – Aggregate – Maximum] or [Utilities – Aggregate – Maximum - String]. The second type of inputs is the running inputs. This input consists of actual prompting messages that will appear in the generated program so that, when run, the program will be able to ask the end user for additional details. One example might be a pathname and filename for the input file of the program.

```
<UserInputs>
    <GenerationInputs>
        <Input RefineGoal="[Utilities -
                Aggregate - Property]"
                Prompt="Which aggregate function
                (Max, Min, Avg, Total)?"/>
        <Input RefineGoal="[IO - Out]"
                Prompt="Where to output (Console,
                File)?"/>
   </GenerationInputs>
   <RunningInputs>
        <Input Name="AggregateUserInput1"
                Prompt="What is your data file?"/>
        <Input Name="AggregateUserInput2"
                Prompt="What is the name of the
                property you would like to
                aggregate?"/>
        </RunningInputs>
</UserInputs>
```

Figure 1: Example Input Portion of a Plan

The heart of a plan is the list of plan steps. Figure 2 illustrates two plan steps of an aggregate plan. The first plan step is used to declare a variable. In this case, the variable is a collection of maps. The second plan step performs an aggregate computation operation on a declared collection. Notice how the type of operation is not specified. This piece of information is required before a specific piece of code can be generated, and the type of operation is obtained via the user specification.

```
<Step Name="records" StepType="Input">
        <Description>
                Declare a collection.
        </Description>
        <Goals>
        [Variables - Declaration - Declare -
                Collection - Of Maps]
        </Goals>
</Step>

<Step Collection="records"
            PropertyName="AggregateUserInput2">
        <Description>
                Apply an aggregate to a
                collection.
        </Description>
        <Goals>
                [Utilities - Aggregate - Property]
        </Goals>
</Step>
```

Figure 2: Two Sample Plan Steps

Now that ACAI has a plan, with its steps, ACAI must locate code segments to fulfill each of the plan step goals. ACAI contains a library of Java code components. The code components come in two different forms. First are fully written methods, each available to handle a type of goal or situation (e.g., an input routine, a sort routine, a search routine). Second are inline or partial pieces of code. These include, for instance, variable declarations, method calls, control statements and assignment statements. All code components are indexed in a similar strategy to ICRT's attributes. In this case, code indexes are described by:

- Type: variables, collections, I/O, control flow, utilities
- Function: declaration (for variables), filter, aggregate operation, event, input/output, assignment statement
- Operation: initialization, criteria for filtering or sorting, type of loop, duration of event, location of input or output
- Data type operated upon

As noted above, every step of a plan is described by a list of goals. Every goal is a generic version of information that can be found in the component library. For instance, a goal might be to declare a collection type of variable. The goal might be expressed as [Variables - Declaration - Declare - ArrayList]. Code components are selected based on how well they match the goal list of the plan step. Additionally, user specifications that include, for instance, whether speed or space is more critical, help select between matching code segments.

Three example code components are listed here. First is an inline statement that declares a collection and initializes it. Notice the use of ^^ symbols. When surrounding an item, these symbols represent a placeholder to be filled in later.

- Component index: [Variables – Declaration – Initialize and Declare – ArrayList]
- Component: Java.Util.ArrayList ^^Name^^ = new ArrayList( );
- Type of component: inline declaration

Second is another inline statement, in this case a loop. Notice the use of placeholders to flesh out the portions of the for-loop that depend on user specifications, such as data type, or an already generated identifier name that replaced a previous placeholder. Replacing placeholders is described below.

- Component index: [Utilities – Iteration – Collection - Map]
- Component: for (java.util.Map <String,String> ^^CurrentItem^^ : ^^Collection^^) { ^^Body^^ }
- Type of component: inline code

Third is a method to compute event duration. Only the header is shown here.

- Component index: [Utilities – Event –Duration – Find Even Duration – int]
- Component: int findEvenDuration (^^StartTimeStamp^^ ^^Name^^);
- Type of component: method

Now, ACAI replaces the component placeholders to construct final component code. In some cases, placeholders represent data types. The selected data type then is used for all matching placeholders. In other cases, names must be generated. For instance, parameter names for methods and variable names replace placeholders. Similarly, method names and method calls must match. ACAI fills in the placeholders and adds the names to complete the component code.

Once ACAI has complete component code, the next step is to fill in the program template. The template comes with the necessary code to make up a Java program. For instance, the template contains proper import statements, a main method, try and catch blocks, as well as additional placeholders.

Another step, which will not be described in detail here, occurs when multiple plans were initially selected. Recall that ACAI contains only a few basic plans. For a simple problem, only one plan would be retrieved. For instance, if the user requires a program to simply sort a collection of patient records by age, only the sorting plan will be required. However, a more complicated problem might involve first filtering records to find patients that meet a particular criterion (e.g., a diagnosis or arrival time), an aggregate computation involving length of stay between events, and finally a sort. Such a problem would require three different plans. In such a case, ACAI would have to combine the three selected plans together. To date, ACAI has only performed modest forms of plan combination.

The code generation process carried out by ACAI results in a program that fits the user specifications to solve the selected problem. Aside from the generated program, if plan combination was performed, the new plan is indexed and stored for future use.

In summary, ACAI uses CBR to retrieve a solution plan. The system uses RD to select appropriate code components and generate the concrete plan steps required to solve the problem. ACAI uses TBP in that it uses a template of a Java program, filling in the details and replacing the placeholders. The overall architecture for ACAI is shown in figure 3.

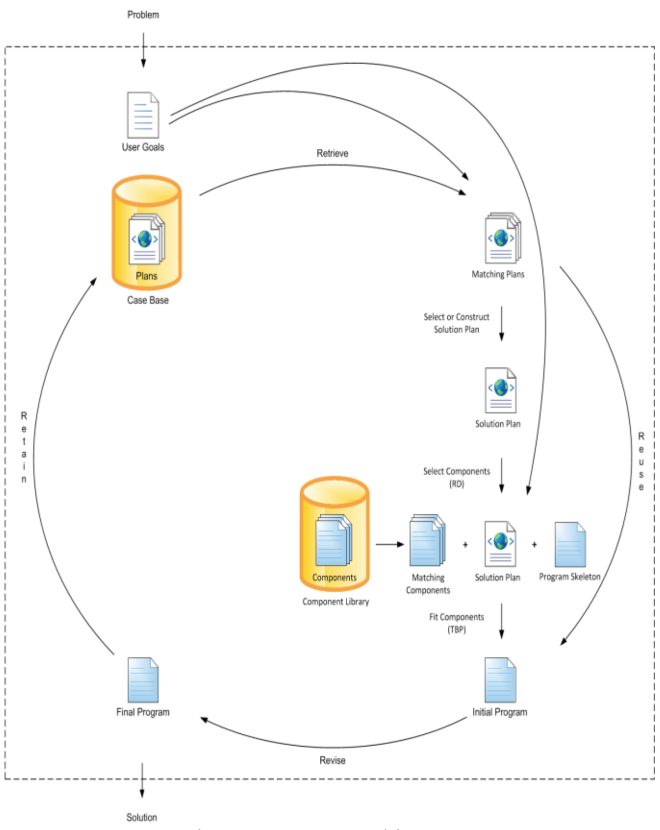

Figure 3: ACAI Architecture

A Brief Example

Here, a brief example is presented to demonstrate how ACAI carries out its code generation task. The user has specified a goal of sorting over integer data and requested the output to be sent directly to the console. Further, the user specifies that speed is of a greater concern than memory space usage.

Based on the input, ACAI retrieves the sort plan. The sort plan contains generation inputs and running inputs. These help specialize some of the goals in the plan steps and provide end user with prompts. The plan steps consist of a declaration of the input collection, an assignment statement to assign a variable to the source of input, a declaration of the sort operation collection variable, a sort routine, and an output step. The goals of these steps are listed here:

- Declare Input Collection: [Variables – Declaration – Declare – Collection – Of Maps – ArrayList]
- Store Input: [Variables – Assignment]
- Obtain Input: [IO – In – File – ArrayList]
- Declare Sorted Data Collection: [Variables – Declare – Declare – ArrayList]
- Store Sorted Data: [Variables – Assignment]
- Sort Data: [Utilities – Sort – ArrayList]
- Output Sorted Data: [IO – Out]

Now, ACAI must identify code components for each of the steps listed above and insert them into appropriate locations of the program template. The template is shown in figure 4.

```
package edu.nku.informatics.thesis.acai;

^^Program Comments^^

public class ProgramSkeleton
{
        public static void main ( String [] args
)
        {
                ^^User Inputs^^

                ^^User Prompts^^

                getUserInputs(userInputs,
                userPrompts);

                ^^Inline Code^^
        }


        // Get the inputs from the user

        ^^Method Code^^
}
```

Figure 4: The Java Program Template

The first code component sought is that of the declaration of input. ACAI selects the following inline statement:

java.util.ArrayList <java.util.Map <String, String>> ^^Name^^;

Here, ^^Name^^ is a placeholder. ACAI now specializes the instruction to the given program by replacing the placeholder with an actual identifier:

java.util.ArrayList <java.util.Map <String, String>> records;

The inline code above is inserted into the template under the ^^Inline Code^^ placeholder. As ACAI continues to find code components to fulfill the given plan step goals, the inline code (whether declaration, assignment or method call) are inserted in order based on the original list of plan steps.

With the identifier *records* in place in the program, ACAI will continue to use this name whenever it must replace other placeholders that reference this same datum.

For instance, the first assignment statement step is handled by the inline code:

^^Variable^^ = ^^Body^^;

which becomes

records = ^^Body^^;

The placeholder ^^Body^^ will be replaced by a method call which will obtain the input and return it as an ArrayList to be stored in records. In this case, the selected method is named readCSVFileIntoArrayList, which contains the code to read data from a file and return it as an ArrayList. This method call is used to replace ^^Body^^.

In many cases, the choice of code component to fulfill a plan step goal is a one-to-one mapping. That is, at least presently, there are few options because of the limited domain that ACAI is working in. However, there are some component options. For instance, there are several different sort routines available. For ACAI to select the best code component for the given goal, user specifications may come into play.

The sort step of this example could be fulfilled by any of six different sort methods. The sort code breaks down into two dimensions: the data type to be sorted and the sorting algorithm. Data types are restricted to numeric, date and string. Since numeric types can be handled generically in Java, Float, Integer, and Double are all sorted by the same routine. As a different type of operation is required to compare two Date objects or two String objects, there is a need for three distinct sorting methods. There are currently two sorting algorithms used in ACAI, Quick Sort and Selection Sort. As Quick Sort uses more space but is guaranteed to be as fast as or faster than Selection Sort, the user specification of speed over memory space causes ACAI to select Quick Sort in this example. The result is that the plan step goal is fulfilled by the following method call:

quickSortNumbers(^^Source^^, ^^Criteria^^);

The ^^Source^^ placeholder is replaced by the aforementioned records variable. The ^^Criteria^^ placeholder references the need for the program to obtain from the end user the criteria by which the sort should operate. This will be the type of data to be compared (e.g., a test result, patient's age, number of visits). The placeholder is replaced by code generated based on the running input. The following is the method call inserted into the program.

quickSortNumbers(records,
userInputs.get("SortUserInput2"));

The program's methods must also be inserted into the template. Methods are largely self-contained and require little change. However, they also contain placeholders, such as variable types, identifier names, and other method calls. The example from this section called for output to console. Assume instead that the output was to be sent to a disk file. Figure 5 contains the stored method selected by ACAI for such an output plan step. Recall that the plan step has the generic goal of [IO – Out]. This must be specialized to fit the user specifications, output to disk file. The ^^Data^^ placeholder in the method call must be replaced with the proper value. In this case, ^^Data^^ becomes list.

Once methods are put into place, the program is complete. ACAI now provides the program as output. An end user can now run the program to solve the desired problem. Running inputs are used to obtain the run-time information required for the program to fulfill the given task.

```
printDataToFile ( ^^Data^^ );


private static void printDataToFile ( Object
objData )
{
    try
    {
        // Declare variables
        java.io.BufferedWriter out = new
        java.io.BufferedWriter ( new
        java.io.FileWriter( "Data.txt"));
        // Write the specified string to the file
        out.write ( objData.toString() );
        // Flushes and closes the stream
        out.close ( );
        System.out.print("Result is stored in: "
           +  System.getProperty("user.dir"));
    }
    catch ( java.io.IOException e )
    {
        e.printStackTrace ( );
    }
}
```

Figure 5: Sample Method Call and Method for Output

Conclusions

ACAI, Automated Coder using Artificial Intelligence, combines the case base, case selection and case storage of CBR with plan decomposition of RD to fill in a template program using TBP. In this case, ACAI succeeds in automated code generation (ACG). Unlike other attempts at ACG, ACAI operates without human intervention other than high level input specification.

In ACAI, plans represent generic solutions to given database type problems. Each plan describes its solution through plan steps. A plan step describes the action required in terms of a goal. Goals provide such information as the type of operation, specific criteria for the operation, and data types.

Given a plan with plan steps, ACAI then selects specific code components from a separate code library. Code components are themselves indexed using attribute lists which match or overlap the goals from plan steps. These code components combine both inline Java code and Java methods. The code components are inserted into a Java program template. Placeholders in the code are replaced by specific identifiers, types, method calls and other programming units as needed.

In order to provide variability, each plan tackles a specific type of operation, such as sort or search. In complex problems, multiple plans are selected and refined into a single solution plan. Plan merging, although not discussed here, provides a seamless transition from one plan to another. The result is a new, more complex plan, which is stored back into the case base for future use.

ACAI has successfully generated programs to solve a number of medical database domain queries and subqueries from the list given in Section 3. ACAI is currently limited to the domain of medical record queries. Although this overly restricts ACAI's abilities, it is felt that the approach is amenable to a wide variety of problems.

It is important to note that the advantage of using ACAI, as oppose to solving the same medical record queries using SQL, is that ACAI's architecture is not restricted to any specific programming language. The ACAI system can be used to tackle a much wider range of problems that would be difficult or inappropriate to address with SQL. Additionally, ACAI allows end users with no programming knowledge to obtain desired results, while SQL would require learning the SQL language as well as having knowledge of programming concepts to accomplish the same task.

Due to ACAI's expandable architecture, theoretically, the only limitation of applying the system in other domains is the availability of associated plans and code components. All that is required to expand ACAI is a greater variety of plans and code components that can implement any new plan steps. Expanding ACAI is a direction for future research along with an examination of additional forms of plan step merging and case reusability. Another direction for future research is increasing the number of criteria that a user might specify for code selection beyond the speed versus space tradeoff mentioned here.

References

Brown, D. C, and Chandrasekaran, B. 1989. *Design Problem Solving: Knowledge Structures and Control Strategies*. Research Notes in Artificial Intelligence Series, Morgan Kaufmann Publishers, Inc.

Brown, D. C. 1996. Knowledge Compilation in Routine Design Problem-solving Systems, *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, p. 137-138, Cambridge University Press.

Chandrasekaran, B, and Josephson, J. R. 2000. Function in device Representation, *Engineering with Computers*, 162-177, Springer.

Danilchenko, Y. (2011). Automated Code Generation Using Artificial Intelligence. M.S. thesis, Dept. of Computer Science, Northern Kentucky University, Highland Heights, KY.

Fahmi, S. A. and Choi H. 2009. A Study on Software Component Selection Methods, in *Proceedings of the 11th international conference on Advanced Communication Technology*, p. 288-292, Gangwon-Do, South Korea.

Fernandez, M. F., Kernighan, B. W., and Schryer, N. L. 1993. Template-driven Interfaces for Numerical Subroutines, in *ACM Transactions on Mathematical Software (TOMS) TOMS p.* 265-287.

Fouqut, G., and Matwin, S. 1993. Compositional Software Reuse with Case-based Reasoning, in *9th Conference on Artificial Intelligence for Applications.* P. 128-134, IEEE Computer Society Press.

Fox, R. and Cox, M. 2000. Routine Decision Making Applied to Nutritional Meal Planning, in the *Proceedings of the International Conference on Artificial Intelligence, IC-AI'2000*, Volume II, p. 987-993, H. R. Arabnia editor, CSREA Press.

Goel, A., Bhatta, S., and Stroulia, E. 1997. Kritik: An Early Case-based Design System, in *Issues and Applications of Case-Based Reasoning in Design*, by M Maher and P Pu, 87-132. Mahwah, NJ: Erlbaum.

Hammond, K. J. 1986. CHEF: A Model of Case-based Planning, in *Proceedings of the Fifth National Conference on Artificial Intelligence*, p 267-271, AAAI.

Hsieh, M., and Tempero, E. 2006. Supporting Software Reuse by the Individual Programmer, in *Proceedings of the 29th Australasian Computer Science Conference*, p 25-33, Australian Computer Society, Inc.

Jiang, Y., and Dong, H. 2008. A Template-based E-commence Website Builder for SMEs, in *Proceedings of the 2008 Second International Conference on Future Generation Communication and Networking Symposia - Volume 01.* IEEE Computer Society.

Jones, C. 2010. *Software Engineering Best Practices.* The McGraw-Hill Companies.

Joshi, S. R., and McMillan, W. W. 1996. Case Based Reasoning Approach to Creating User Interface Components, in *Proceedings CHI '96 Conference companion on Human factors in computing systems: common ground*, p. 81-82.

Vazquez, G., Pace, J., and Campo M. 2008. A Case-based Reasoning Approach for Materializing Software Architectures onto Object-oriented Designs, in *Proceeding SAC '08 Proceedings of the 2008 ACM symposium on Applied Computing*, p 842-843, ACM.