CONVERSION OF EXECUTABLE DESCARTES SPECIFICATIONS INTO PROGRAM CODE

Joseph E. Urban  
Industrial Engineering Dept  
Texas Tech University  
Lubbock, Texas 79409 USA  
joseph.urban@ttu.edu

Praneet K. Rattan  
Software Engineer  
Caradigm  
Murray, Utah 84107 USA  
rattan.praneet@gmail.com

Vinita H. Subburaj  
Computer Science Dept  
Texas Tech University  
Lubbock, Texas 79409 USA  
vinita.subburaj@ttu.edu

Austin E. Carver  
Physics and Astronomy Dept  
University of Louisville  
Louisville, KY 40292 USA  
alcarv01@louisville.edu

ABSTRACT
Automatic programming is the systematic generation of a program from a given specification. The goal of automatic programming is to allow programmers to specify what a program should do, and let the system generate the program code describing how the given program will work. Programs can be specified using a constructible, formal, and comprehensible specification language. Descartes is one such formal specification language, based on the functional model and has the advantages of easy constructability and comprehensibility. This paper overviews the use of formal methods to specify requirements and the advantage of using an executable formal specification language in order to specify software systems. In Descartes, specifications are described by defining the input and output data and relating the output data as a function of the input data. This research paper analyzes the various approaches towards automatic programming and reduces the gap between specification and programming, by introducing a method to automatically transform a Descartes specification of a program into executable Java code.

KEY WORDS
automatic programming; formal specification; the Descartes; program code

1. Introduction

The software life cycle is a process consisting of formal and logical steps: such as planning, requirements analysis, design, programming, testing and maintenance, taken to develop a software product. Requirements analysis is a significant phase of the software life cycle because if any fault in the specification is left undetected, it will be carried over into the next phase. Thus, later correction of the fault would involve fixing the fault itself and fixing the effects of the fault in subsequent phases. Schach [1] illustrates that the relative cost to detect and correct a fault is more during the programming and maintenance phase, as compared to other phases. For example, Faulk [22] in his paper discussed the relative cost to repair a software error during requirements stage is 1-2 when compared to the coding phase, which is approximately equal to 10. Apart from consumption of a large amount of resources, for a change in a specification, the entire code will have to be rewritten by the developers. Also, since there is no concrete system to verify the requirements provided by the user in a conventional software development life cycle, development of such unverified requirements could cause errors in the program code.

Automatic programming is a mechanism, which generates a computer program from specifications written by human software developers at a higher level of abstraction. Automatic programming eliminates the design and coding phases from the software life cycle and conserves resources, which can be used for other processes. Program developers could potentially save time and effort through automatic programming. For the slightest change in specification, the developers do not have to rewrite the entire code, but can input an updated specification into the system to automatically update the code.

The efficiency of designing an automatic system widely depends on the specification language used. The specification language must be user-friendly, easy to understand, constructible, formal, and applicable to various problems. Descartes is a formal specification language, evaluated with respect to the criteria established by Liskov and Zilles [2], which includes formality, constructibility, comprehensibility, minimality, wide range of applicability, and extensibility. Descartes is based on the functional model and the specifications are described by defining the input data as a function of the output data [12, 19, and 21]. In the research effort, an analysis of possible approaches towards implementing automatic programming has been made. Using Descartes as the specification language, a system has been designed which analyzes the specification and then produces the corresponding program code.

The field of automatic programming is vast and yet not completely explored. Several approaches are followed to implement an automatic program generator. In this research effort, most of the existing techniques related to automatic generation of program code were studied in detail. However, the hybrid combination of two or more known techniques was found to be more efficient than the existing techniques, and hence was further analyzed. Since the research effort is concentrated in the domain of arithmetic programs, various elements of arithmetic problems and their role and effects in the real world were identified and
analyzed. For example, in order to construct a program to find the minimum from a given set of numbers, elements such as the set of input numbers, type of given numbers, and functions to be applied on the given numbers were analyzed prior to the implementation of the respective program.

Based on the results of the analysis on various arithmetic elements, a library of arithmetic functions was created. The library consists of basic arithmetic functions and their definitions. The definitions of arithmetic functions were written in the target programming language, Java. Apart from the functions present in the library, prior to the use of any other arithmetic function in the specification, the function definition needs to be manually added to the library. In order to transform a specification into program code, it is essential to understand the representation of data structures used by the specification language and identify the semantics used by the target programming language for representing such data structures. Hence, a diverse set of specifications from the domain of arithmetic problems were selected and represented in the target programming language. By analyzing the selected set of programs, a correlation was established between the data structures used by the specification and the target programming language. Once the function library and the common structure for program were defined, a parser was developed to parse through any related Descartes specification. The parser was developed in Java and uses the function library for definition of arithmetic functions used by a specification. The parser analyzes a Descartes specification and transforms it into Java code. Such parsing undergoes multiple transformations until the target program code is obtained. After development, the parser was tested using a wide set of specifications and evaluated.

Arithmetic is one of the strong foundations of computer science. Arithmetic problems range from simple day to day counting to advanced science and business calculations. Even large arithmetic problems can be decomposed into sub problems, where each sub problem is a simple calculation. Since Descartes is a formal and constructible language, formulating an advanced problem as a simple specification using basic arithmetic functions is not difficult. Hence, in this research effort, only specifications for programs from the domain of arithmetic problems were considered as input. Development of such systems in other fields of programming problems can be carried out on similar lines, as future work.

The remainder of this paper has Section 2 that is related work on previous research efforts that influenced this research project. Section 3 describes the role of executable specifications. Sections 4 and 5 describe the code generation system that was developed for the Descartes specifications. Section 6 discusses the evaluation results of the proposed system. Section 7 provides an overview of an approach to combining the reverse engineering capability with this system. Section 8 includes a summary and future research.

2. Related Work

The history of automatic programming can be traced back to the 1940s where the term was used to describe automation of the manual process of punching paper tape [3]. The 1950s witnessed the foundation of early FORTRAN compilers and in the 1960s and 1970s fields such as compiler theory, formal methods, and artificial intelligence contributed to the foundations of automatic programming. Though several research efforts have been made, each used a different approach to achieve automatic programming.

Jazayeri [4] proposed a system that can be used to automatically produce text-processing programs using specifications. Jazayeri noted that the specification language should be chosen as suited to the type of programming task. Knuth’s Attribute Grammar was identified as a good specification language for programs involving text processing. An attribute grammar is a formal way of defining attributes for the production of a formal grammar. Using a parser, the input grammar is parsed and an abstract syntax tree is produced. Each node of the abstract syntax tree is associated with an attribute. A program that implements the attribute grammar is called an Evaluator. Jazayeri uses the k-ASE class of evaluators for an evaluation grammar. Jazayeri [5] has also stated an algorithm, which determines if a given grammar belongs to the K-ASE class. The algorithm also produces sets of attributes $A_i$, evaluable at pass $i$. The information about attributes is incorporated into a parser to obtain an efficient evaluator.

Xinger [6] proposed a technique for converting a specification directly into a program by using source-to-source conversion. The proposed system is based on formal rules and Canonical Abstract Syntax Trees (CAST). A CAST is a tree representation of simplified structure of source code written in a programming language, where a subtree can be evaluated in any order. The system supports rapid prototyping; and executable specifications are an effective approach to rapid prototyping. The system uses CAST as an immediate form of representation. The key to source-to-source conversion is to build the corresponding relationship between the two languages. A formal specification language possesses fundamental features such as syntax, semantics, and pragmatics. By finding out the relationship between the fundamentals of two different languages, a program specification $P_A$ can be converted to be a program in programming language $A$, $P_A$.

The specification language used for such conversion should be executable, modular, formalized, logical, clear, and unambiguous and have good expressive power. Considering these factors, Xinger designed a language called NCISL, as a suitable specification language. NCISL describes the functionality of a system by explaining the effect of execution and not by using equations, which express the relationship between functions. Xinger [6] states that the system developer needs to ensure that structures of both specification and programming language comply with each other.
Specification languages can be modified, extended, and used in automatic programming systems built for specific application-based requirements. Jacob [7] proposed a specification technique based on state transition diagrams and uses an interpreter, which executes formal specifications to build a user interface. The user interface is divided into three levels, i.e., semantic, syntactic, and lexical levels. Each of these levels has a separate executable specification. For the system developed by Jacob, the specification interpreter is written in C and a common front end is constructed using YACC and LEX, which is used to parse the specification, both for interpreting it and converting it to diagram form.

In recent years, many methods have been employed on similar lines to automatically generate program code from specifications. Arcuri and Yao [8] used genetic programming to evolve programs from their specifications. The paper introduces a framework where the goal of automatically refining specifications is reached by evolution of programs and unit tests. Shirakawa et al. [9] introduced a new method called Dynamic Ant Programming, using Ant Colony Optimization for automatic generation of programs and compared the approach to genetic programming.

3. Executable Specifications

Specification has an important role during software development. At times, software engineers while developing software systems can overlook the importance of the specification stage. Software life cycle models are improvised for better software quality and also reliability. Even though powerful implementation strategies exist, if the requirements of a system to be developed are unclear, and not specified properly then the product will fail to satisfy user expectations. Specifications that serve as a communication tool between the user and the developer is still less sophisticated due to the boundary line that exists between formal and informal descriptions of the system to be developed. The use of executable specifications remains a challenging research problem that needs to be addressed. Fromherz [10] addressed that an executable specification represents not only a conceptual, but also a behavioral model of the software system to be implemented.

Formal methods in specification languages have used mathematical techniques to specify software systems. Formal methods result in delivering reliable software when applied early during software specification stages. In this paper, one such formal specification language that is executable was under study. Formal specification uses well defined syntax and semantics to specify the software system that can be transformed to an executable version of the specifications. Executable specifications will ensure that the requirements gathered and listed for the system demonstrate the actual behavior of the system that is efficient and reliable for implementation [13, 14].

Figure 1 is an example of a Descartes specification that performs the reversal of a string of integers. The STRING analysis tree is followed by the return synthesis tree.

4. Functionalities of the Developed System

Urban [10] developed a system capable of executing Descartes specifications called the Descartes language processor. After a Descartes specification has been composed, the Descartes language processor can be used to attest the specification for correctness, completeness, and consistency with the intentions of the software developer. The composed specification is correct if for a given set of correct inputs, the specification executed by the Descartes language processor produces correct output as expected. The system developed during this research effort, called AUTOCODE (AUTOMATIC CODE generation from a DESCartes specification), accepts a Descartes specification for a given program and produces the corresponding program code in Java.

AUTOCODE serves as an extension to the Descartes language processor. While the Descartes language processor checks for consistency and completeness of a Descartes specification; AUTOCODE scans, analyzes, and interprets the specification and produces the corresponding program code in Java. However, the current AUTOCODE system can only work on specifications from the domain of arithmetic problems.

4.1 Scanning and Interpretation

The Descartes specification language is based on the functional model. A Descartes specification is described by defining the input data as a function of the output data. In order to implement the functional relation between the input and the output data, properties of a problem; such as type of input parameters, structuring of data, association between different parameters, must be known. Such properties help in understanding the problem and structuring a suitable solution. A complex problem can be structured as a Descartes specification by using a combination of pre-
4.2 Storing Data

Once the properties of a specification are known, the related data is stored in a database. The database for every specification is custom designed to suit the needs of the specification. Designing of the database is carried out at runtime, and hence the database can model itself according to the specification. In addition to the database, an XML file is created for each specification. While the database stores detailed information about each node in the specification, the XML file stores general information about the structure and relationship between nodes. The stored data can be directly accessed and can also be used for debugging.

4.3 Producing Program Code

A program code generator for producing the program code can access the data from the database and the XML file. The program code generator uses the data models to build a relationship between the Descartes specification language and the target programming language. In other words, for any correctDescartes specification $S_D$, the program code generator generates the corresponding program code $P_T$, such that $S_D = P_T$. Execution of the generated program code yields the same results as the execution of the Descartes specification.

4.4 Documentation

The AUTOCODE system records the properties and structure of the specification in the respective XML document. In addition to the XML document, AUTOCODE documents functions, attributes, and properties of the program in the generated code itself, as comments. By doing so, AUTOCODE makes the generated code self-explanatory.

5. System Modules

This section explains the different modules of the AUTOCODE system, their working and functions. AUTOCODE can be logically divided into five modules: specification scanner, interpreter, data storage, code generator, and function library.

Figure 2 explains the logical division of the system. A Descartes specification is read by the specification scanner, which extracts the preliminary information from the specification and stores the information in the database. The specification scanner also synthesizes the Descartes specification into two parts: structural and executable. The interpreter performs analysis on the specification, extracts modular chunks of data, stores the data in the database, and creates an XML file explaining the structure of the specification. The data stored in the database and XML file are accessible by the code generator. The code generator analyzes the data, relates it to the programming language, and produces the corresponding program code. The code generator has access to a function library, which stores primitive functions capable of performing arithmetic, logical, and Boolean operations.

5.1 Specification Scanner

A Descartes specification is constructed using Hoare trees [11]. The specification is developed in a top-down manner and is divided into modular units that refine the Hoare tree until no further refinement is desired or possible. A Descartes specification can have one or more modules. A module may or may not be dependent on other modules in the specification.

A module begins with a natural language phrase in uppercase letters, called a title. The title describes the function of the module and asserts the arguments. The arguments are in uppercase letters and enclosed in parentheses. Each module consists of two types of trees: analysis tree and synthesis tree. An analysis tree is a Hoare tree describing the pattern of input where the root node is the complete input while each intermediate and terminal node is possibly a part of the input. A synthesis tree is a Hoare tree that describes the output of the module, in terms of the nodes present in the analysis tree. Thus, a module analyzes the input data and synthesizes output as a function of the input data. Considering the development of AUTOCODE, a specification module can have more than one analysis tree, but only one synthesis tree. The value of the module is determined by the synthesis tree that is distinguished from other Hoare trees by the use of “return” as its root node. The terminal nodes of a “return” synthesis tree can be match nodes from the analysis trees in the module, or can be literals, or can be a module call.

The specification scanner divides the specification into two parts: structural and executable. The structural portion of a specification deals with defining the input and its structure. Hence, the structural portion of a specification comprises of the title and all the analysis trees in the specification. The executable portion of a specification explains how the desired output can be achieved by manipulating the input. In other words, the executable portion of a specification explains the mechanism of the
program. The executable portion comprises of the synthesis
tree with a “return” root node. The specification scanner
performs preliminary data analysis on the specification. The
analysis includes identifying node levels and datasets.

5.2 Identifying Node Levels

A Hoare tree is made up of one or more nodes. When the
number of nodes is greater than one, the Hoare tree can
extend to multiple levels where every node has a parent
node, except the root node. A parent node may have more
than one child node, but a child node can have only one
parent node. In a Descartes specification, only one node can
be written on a line. There is always a newline character
between two nodes. The level of a node is identified by the
amount of indentation prior to the node on the same line.
The number of indentations in one line of specification is
directly proportional to the level of the node. The
specification scanner reads each line of the specification,
identifying the nodes, as well as assigning them level values.
For the structural portion of a specification, the specification
scanner identifies the module name, associated arguments,
match nodes associated with the analysis tree of each
argument, terminal literals or reference nodes and their
respective levels. For the executable portion of a
specification, the specification scanner identifies the
reference nodes, terminal literals, module calls, and their
respective levels.

5.3 Identifying Datasets and Reference Nodes

In a Descartes specification, a terminal node of an analysis
tree can either be a reference node, or set of literals, or a
module call. Whenever the specification scanner encounters
a literal or a set of literals as terminal nodes, under the same
parent node and at the same level, it replaces the literal or
set of literals with a new type of node, called dataset, and
the dataset points towards the replaced literals. Introduction
of datasets into the original Descartes system is purely
implementation based. Since referencing a set of literals is
easier than referencing each literal multiple times, the
concept of dataset has been introduced.

A reference node can exist anywhere in the
specification. The reference node could point towards a
match node which may or may not exist in the same analysis
tree, but must exist in the module. However, a reference
node cannot exist in the subtree of its match node, unless it
is unlikely to create an infinite self-loop. The specification
scanner identifies all the reference nodes existing in a
specification. The specification scanner performs
preliminary analysis and stores the results in the data storage
module of the system. The structural and executable
portions of the specification are passed on to the interpreter
for data analysis.

5.4 Data Storage

The data storage module is the central hub for storing any
data concerning the specification. The data storage is
comprised of two units, a database and an XML document.
Both units are accessible by a programmer and hence can be
used for debugging, if needed. However, the XML
document is read-only, but the database can be edited by
any source outside the system.

The database is developed dynamically during runtime.
The data stored in the database can be divided into four
domains: module data, structural data, executable data, and
definition of datasets. For every module in a specification,
module data holds information such as module title,
associated arguments, and basic information about the
arguments. For each module present in the specification, the
structural data consists of information about match nodes
present in that module and associated properties. The XML
documents states the structure of the specification. The
document has attributes, which hold the level of each
number and its corresponding primary key in the database.

5.5 Interpreter

The interpreter performs advanced analysis on the
specification and identifies all properties of a node. The
interpreter determines the associativity between nodes based
on the properties of a node. The interpreter acts as a
mediator by extracting all information from the specification
and distributing it accordingly in the database. The code
generator can directly access information from the database,
design a framework for the program, and fill in program
code corresponding to the functionality of each node.

During the initial stages of the research, a set of
arithmetic problems were studied, composed as Descartes
specifications and implemented in the target programming
language. While selecting the set of arithmetic problems, an
effort was made to include at least one problem from each
domain of arithmetic problems. Also, the set of problems
was selected in such a way that the Descartes specification
for these problems included almost all combinations of data
structures present in the Descartes language. After the
arithmetic problems were implemented, the program code
was studied to identify a set of common attributes for all
nodes, which was helpful in structuring the program. In
addition to the attributes, a common pattern of structural
code was identified for each type of node and data structure
present in the specification. The main characteristic of the
identified common pattern was that the program code for a
combination of nodes using different data structures, where
each node is described using the common pattern, would be
executable and produce correct output on execution. Using
the common pattern for different types of nodes and data
structures, a set of data models was designed. While the
interpreter identifies the values for the set of common
attributes and processes them accordingly, the common data
models are used by the code generator in building the
program and are discussed in the next section.

5.6 Code Generator

The code generator is the main engine of the AUTOCODE
system. The code generator analyzes the information
extracted by the interpreter, builds a framework for the program, and fills in corresponding program code for each node.

5.6.1 Designing Program Framework

For every module in a specification, the code generator creates a class (*.java) for each of the analysis trees present in the module. For every analysis tree, the code generator proceeds incrementally, starting from the root node, followed by subnodes and so on, until all nodes in the tree are processed. If the code generator encounters a terminal node, it backtracks until it finds another unprocessed node. For each node encountered, the code generator creates a node-function.

5.6.2 Node-Functions

In terms of implementation, each node is implemented as a function (node-function) that returns a true or false value. If the node is at level 0, then the input to its node-function is the input parameter obtained from the user. If the node is at any level, other than level 0, then the input to its node-function is derived from its parent node. The attributes of a node are used in structuring the node-function. If the operation of a node is direct product, sequence, or limited sequence, then the node-function would return true provided the node-function for all of its subnodes are true. However, if the operation of a node is a discriminated union then the node-function would return true provided the node-function of only one of its subnodes is true. If a node-function returns true then it implies that the corresponding node is valid and the value of such a valid node is stored. Once all nodes in a specification are processed incrementally from level 0 up until terminal nodes and found to be true, then it implies that the initial input provided by the user too is true and valid.

5.6.3 Function Library

The Descartes specification language has some primitive modules pre-defined in the system. Each primitive module accepts a limited number of arguments where each argument is a reference node. The primitive modules are used to perform arithmetic, logical, and Boolean operations. AUTOCODE stores these modules in a function library. If a module in a specification calls for any of the predefined primitive modules, the program control automatically transfers to the function library, executes the module, and returns the respective output to the calling module. New primitive modules can be added to the library as desired. For the current system, primitives related to only arithmetic functions have been added.

6. Evaluation of the Proposed System

The AUTOCODE system has been evaluated in terms of time and space. To evaluate the system, a set of five programs was chosen. The codes for the programs were generated manually as well as using the AUTOCODE system. While implementing the programs manually, an attempt was made to use algorithms with least number of steps and complexity.

For the set of programs listed in Table 6.1, the code was generated using the AUTOCODE system. For the same set of programs, code was also generated manually. The average time of execution for both, manually generated code and code generated using AUTOCODE was calculated. In order to decrease the error in determining the time of execution, following measures were taken:

- same input was given to each of the two executions of a program;
- the input was hardcoded into the program code, to avoid marginal time;
- difference caused while manually entering the input;
- the time of execution was calculated in nanoseconds to increase accuracy; and
- a program code was executed five times for the same input and an average of the five instances of execution time was considered to tabulate results.

Based on the obtained average time of execution, a chart was plotted, as shown in Figure 6.2

<table>
<thead>
<tr>
<th>Program</th>
<th>Name of the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Display the odd digits present in a given number</td>
</tr>
<tr>
<td>2</td>
<td>To reverse a given string of integers</td>
</tr>
<tr>
<td>3</td>
<td>Enumerate from any given number till 100</td>
</tr>
<tr>
<td>4</td>
<td>Display sum of two numbers</td>
</tr>
<tr>
<td>5</td>
<td>Count the number of digits in an integer</td>
</tr>
</tbody>
</table>

Table 6.1 List of programs used to evaluate the system

After plotting the chart, the following was observed:

- time of execution for a program coded using the AUTOCODE system was almost same as the time of execution for the same program coded manually;
certain programs (program id: 2, 4) which involve usage of recursive or pre-defined functions, exhibit less time of execution when generated using the AUTOCODE system;
the code generated through the AUTOCODE system for program specifications which have no node with a sequence operation in the synthesis tree, execute faster than their counter manual implementation;
the code generated through the AUTOCODE system for program specifications which have a node with sequence or discriminated union operation in the synthesis tree execute slower than their counter manual implementation.

Figure 6.3 Memory used by the programs at runtime

The runtime memory used by each program was also determined and a chart was plotted, as shown in Figure 6.3. The memory usage for manually implemented programs did not show much variation, but the programs generated using AUTOCODES showed a significant amount of change in terms of memory usage, and was dependent on the types of nodes used in the program specification.

7. Matching Programming Constructs and Reverse Engineering

The executable Descartes specifications were analyzed in detail for conversion purpose. Matching of the Descartes specification language constructs with the semantics of a programming language was made initially [15, 16, 17, and 18]. Pre-defined matching scenarios enable the conversion of appropriate specification constructs into implementable code. A tool to convert the specifications written in the extended Descartes specification language into programmable code was developed in this research effort. The Descartes specification language uses Hoare trees for defining the data structuring methods [20]. The three structuring methods used in the Descartes specification language are namely direct product, discriminated union, and sequence. Each of the structuring methods corresponds to programming statements as follows:
1. direct product relates with compound statements in programming languages;
2. discriminated union relates with conditional statements; and
3. sequence relates with iterative statements.

The ability to do a check of fulfilled specifications on recently developed software is a need; as is the need to check that the software contains no malicious code. Reverse engineering programs to its specifications can help to show if the program completely satisfies the intended design. Reverse engineering programs helps with error checking and testing, making the whole process more efficient and less expensive. This research effort focused on developing a method to transform C++ programs into Descartes specifications and to determine how to transform more complex programs into specifications.

After working on designing a program to transform a C++ program into Descartes specifications, the situation became apparent that a compiler is a better tool for creating a Hoare tree as a log file. This design has some assumptions: no pointers are in the program and the software developer uses common programming practices (C++ [ample curly brackets]). This design provides various advantages: the code does not have to be fully developed for this method to be used and provides the software developer and a third party with safeguards and quality checking methods. Yet, this design does have some disadvantages: does not yet read class files, does not fully read pointers because the three data structuring methods eliminate the need for pointers, and simple and short programs would not benefit from this method.

8. Conclusion

This paper discussed the functionality of the AUTOCODE system. An overview of system modules was presented with detailed explanation about each module. The specification scanner aims at extracting preliminary information from the given Descartes specification and preparing the specification for further analysis. The interpreter accepts the modified specification from the specification scanner and performs analysis. The interpreter extracts the attributes associated with each node in the specification and accordingly stores the information in the data storage
module of the system. The code generator uses the information stored in the data storage module, compares it to the set of data models designed during the initial stages of analysis, and correspondingly produces the program code in the target programming language. The code generator also has access to a function library, which stores a set of predefined primitive modules for performing basic arithmetic functions. The AUTOCODE system was developed using the Java programming language and produced the output program code in Java as well. The system was evaluated on the criteria of time, space and algorithmic complexity. After comparing the results obtained from the manual implementation of a specification and the program code generated by AUTOCODE, the performance of program code generated through the AUTOCODE system matched considerably as compared to the performance of manual implementation. Under certain circumstances, the performance of program code generated using the AUTOCODE system, exceeded the performance of manually implemented program. In addition to the advantage of increased performance, using AUTOCODE does eliminate the overhead of programming and debugging. As a result, resources saved in terms of human effort, money and time can be used for better utilization. Also, for any minor change to the specification, the programmer would not have to go through the process of redevelopment, but instead, just generate the new program code for the changed specification.

There are future research efforts planned to expand the scope of primitive modules. In addition, the reverse engineering capability will be expanded to fully support the language features.

Acknowledgement

DISCLAIMER: This material is partially based upon work supported by the National Science Foundation under Grant No. CNS-1005212. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References