December 1988

Reverse Software Engineering

Noah S. Prywes
University of Pennsylvania

X. Ge
University of Pennsylvania

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

M. Song
University of Pennsylvania

Recommended Citation

Noah S. Prywes, X. Ge, Insup Lee, and M. Song, "Reverse Software Engineering", . December 1988.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-99.

Reverse Software Engineering

Abstract

The goal of Reverse Software Engineering is the reuse of old outdated programs in developing new systems which have an enhanced functionality and employ modern programming languages and new computer architectures. Mere transliteration of programs from the source language to the object language does not support enhancing the functionality and the use of newer computer architectures. The main concept in this report is to generate a specification of the source programs in an intermediate nonprocedural, mathematically oriented language. This specification is purely *descriptive* and independent of the notion of the computer. It may serve as the medium for manually improving reliability and expanding functionally. The modified specification can be translated automatically into optimized object programs in the desired new language and for the new platforms.

This report juxtaposes and correlates two classes of computer programming languages: procedural vs. nonprocedural. The nonprocedural languages are also called *rule based, equational, functional or assertive*. Non-procedural languages are noted for the absence of "side effects" and the freeing of a user from "thinking like a computer" when composing or studying a procedural language program. Nonprocedural languages are therefore advantageous for software development and maintenance. Non procedural languages use mathematical semantics and therefore are more suitable for analysis of the correctness and for improving the reliability of software.

The difference in semantics between the two classes of languages centers on the meaning of variables. In a procedural language a variable may be assigned multiple values, while in a nonprocedural language a variable may assume one and only one value. The latter is the same convention as used in mathematics. The translation algorithm presented in this report consists of renaming variables and expanding the logic and control in the procedural program until each variable is assigned one and only one value. The translation into equations can then be performed directly. The source program and object specification are *equivalent* in that there is a one to one equality of values of respective variables.

The specification that results from these transformations is then further simplified to make it easy to learn and understand it when performing maintenance.

The presentation of translation algorithms in this report utilizes FORTRAN as the source language and MODEL as the object language. MODEL is an equational language, where rules are expressed as algebraic equations. MODEL has an effective translation into the object procedural languages PL/1, C and Ada.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-99.

REVERSE SOFTWARE ENGINEERING

N. Prywes, X. Ge,
I. Lee and M. Song

MS-CIS-88-99

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

December 1988

Prepared Under Contract AFOSR-88-0116

from the

Air Force Office of Scientific Research

Bolling Air-Force Base, DC 20332-6448

ABSTRACT

The goal of Reverse Software Engineering is the reuse of old outdated programs in developing new systems which have an enhanced functionality and employ modern programming languages and new computer architectures. Mere transliteration of programs from the source language to the object language does not support enhancing the functionality and the use of newer computer architectures. The main concept in this report is to generate a *specification* of the source programs in an intermediate nonprocedural, mathematically oriented language. This specification is purely *descriptive* and independent of the notion of the computer. It may serve as the medium for manually improving reliability and expanding functionally. The modified specification can be translated automatically into optimized object programs in the desired new language and for the new platforms.

This report juxtaposes and correlates two classes of computer programming languages: procedural vs. nonprocedural. The nonprocedural languages are also called *rule based, equational, functional or assertive.* Non-procedural languages are noted for the absence of "side effects" and the freeing of a user from "thinking like a computer" when composing or studying a procedural language program. Nonprocedural languages are therefore advantageous for software development and maintenance. Non procedural languages use mathematical semantics and therefore are more suitable for analysis of the correctness and for improving the reliability of software.

The difference in semantics between the two classes of languages centers on the meaning of variables. In a procedural language a variable may be assigned multiple values, while in a nonprocedural language a variable may assume one and only one value. The latter is the

same convention as used in mathematics. The translation algorithm presented in this report consists of renaming variables and expanding the logic and control in the procedural program until each variable is assigned one and only one value. The translation into equations can then be performed directly. The source program and object specification are *equivalent* in that there is a one to one equality of values of respective variables.

The specification that results from these transformations is then further simplified to make it easy to learn and understand it when performing maintenance.

The presentation of translation algorithms in this report utilizes FORTRAN as the source language and MODEL as the object language. MODEL is an equational language, where rules are expressed as algebraic equations. MODEL has an effective translation into the object procedural languages PL/1, C and Ada.

Contents

List of Figures

List of Tables

1. INTRODUCTION AND SUMMARY

Much research and development has been directed in the past to the overall Reverse Software Engineering problem: i.e. how to utilize outdated programs to reduce cost of developing new replacement systems. There are existent systems for "structuring" programs to make them more readable and understandable [6,12]. There are also systems that transliterate from one procedural language to another [10,20,21,22,23,33,37]. The approach described in this report differs in a number of ways. It generates a computer independent mathematical meaning of the program. The abstract explanation of the program supports analysis of its correctness and serves as the medium in which maintenance of the program can be performed. The intermediate nonprocedural language is shown at the center of Figure 1. It is useful for both, new software development and for software maintenance and updating. In this report we will refer to this nonprocedural language as *equational* language. We will call the input to the translator *program* and the output *specification.*

This report focuses on the Reverse Software Engineering translation shown at the top half of Figure 1. The objective of this report is to present the algorithm that translates the procedural program into the equivalent of mathematical equations. The coupling with the Forward Software Engineering, shown at the bottom of Figure 1, allows to obtain automatically programs in a new programming language and for a new computer architecture.

This report juxtaposes and correlates the two classes of computer programming languages: conventional *procedural* languages vs. the more recently introduced nonprocedural mathematically oriented languages. The latter class has been called *rule based[8,40], equational,* [4,30,35] *functional*[5] *dataflow* or *assertive* [1,29]. Procedural languages are *prescrip-*

tive and consist of an ordered set of statements that contain commands to a computer. Two major difficulties with procedural languages have been widely recognized: the need for the user to "think like a computer" in composing or reading a program, and the existence of "side effects" that make it difficult to understand the meaning or modify any one statement, as it is frequently effected by other statements [4,7]. The nonprocedural languages are purely *declarative* or *descriptive*. Each statement is a description of a stand-alone mathematical rule applied to entities in the program requirement. Order of statements can be immaterial. There are no side effects. The meaning of each statement can be understood without even mentioning any computer concepts. It has been widely claimed that the nonprocedural class of languages is much superior for computer programming[5,8,13]. The adoption of languages of this class will greatly reduce cost of software development and maintenance and improve reliability of the produced programs. This class of languages can be effectively translated into procedural programs[14], as shown in the lower half of Figure 1.

```
                              +------------------------+
                              |                        |
                            +------------------------+ |
                            |                        | |
                          +------------------------+ | |
                          |        Existing        | |-+
                          |   Procedural Programs  |-+
                          +------------------------+
                                       |
                                       |
                                       V
                          +------------------------+
                          |      Translation       |
                          |      -----------       |
                         (Reverse Software Engineering)
                          |                        |
                          +------------------------+
                                       |
                                       |
                                       V
                              +------------------------+
                              |                        |
                            +------------------------+ |
                            |                        | |
                          +------------------------+ | |
    O                     |          New           | | |
    |      /|             | Equational Specification| |-+
   /|\    / | <---------------->|                  |-+
    | ___/  |   Compose   +------------------------+
   / \|_____|   read                   |
                modify                 |
                verify                 V
                          +------------------------+
                          |      Translation       |
                          |      -----------       |
                         (Forward Software Engineering)
                          |                        |
                          +------------------------+
                                       |
                                       |
                                       V
                          +------------------------+
                          |    New or Improved     |
                          |   Procedural Programs  |
                          +------------------------+
```

Figure 1: Use of Translations Between Procedural and Nonprocedural Languages

There have been a number of theoretical research directions that involve giving a mathematical meaning to a program as follows. Verification of correctness of programs has involved giving a mathematical meaning to a program, and using the latter representation in proving correctness [9,17,19,24]. It is much easier to conduct the proof on the mathematical representation of the program. Research on program transformations has utilized a similar approach making it easier to perform the transformations [7,11,31]. Compiler generation has been based on using mathematical definitions of a programming language in the form of its denotational semantics [18,32,38]. The translation from sequential into parallel programs [2,25] utilizes an intermediate assertive language. The mathematical representation developed in this report is in the form of regular and boolean equations. It is much more widely familiar and more readily manipulateable. It potentially can be used for the above purposes as well.

The basic difference between procedural and equational languages is in the meaning of variables. Procedural languages allow multiple assignments to a variable in the course of executing a program. In contrast, an independent variable in mathematics can assume only one value. The translation algorithm in this report incorporates transformations that progressively rename the instances of assignments to a variable, until single value assignments are attained throughout. The translation to equations is then directly attainable. There is *equivalence* between the source program and the object equations in that corresponding variables have the same values, respectively.

The choice of languages for presenting and illustrating the translation is as follows. We chose FORTRAN[15] as an example of a procedural language because of its relative simplicity in comparison with other procedural languages. We chose MODEL[13,34] as the example of an equational language because of its equational syntax and semantics and the highly developed state of its Forward Software Engineering translation. Presently, translators are availablė from MODEL into PL/1, C and Ada procedural languages[14]. The existent MODEL Forward Software Engineering translation algorithms have provided the basic ideas for the Reverse Software Engineering translation algorithms.

In the following we will assume that the source procedural language program has been pre-processed into basic types of statements that are common to the entire class of procedural languages. Also the MODEL language includes only the basic types of statements for declarations of variables and expressing rules as regular and boolean algebraic equations. The equations may have to be post-processed into other nonprocedural languages. Thus the choice of the languages in this report should not impose a restriction on the basic capabilities of the translation from procedural to non procedural languages.

In addition to this introductory section, the report consists of four sections. Section 2 describes the overall approach to the translation problem. Further insight into the approach is provided in Section 3 through presenting the translation algorithm and by illustrating it with an example of the translation of a small program into a respective equational specification. The objective is to give the reader an understanding of the translation through this example. Additional translation issues are discussed in Section 4. The report ends with a concluding section 5.

2. THE OVERALL APPROACH TO TRANSLATION FROM A PROCEDURAL LANGUAGE TO AN EQUATIONAL LANGUAGE

2.1 Overview

This section describes the basic ideas that underlie the translation.

Given a source procedural program, it will first be translated into a program that uses a basic subset of FORTRAN statements. This basic subset is generally common to all procedural languages. It is briefly described in section 2.2. This is followed in section 2.3 by description of the basic part of the MODEL *equational specification* language, which is the object of the translation. The basic MODEL language concepts are presented only briefly in this section, and further discussed and illustrated in section 3. A brief review of the transformations in the translation process is given in section 2.4.

The objective of the translation is to attain *equivalence* between the source (FORTRAN) program and object (MODEL) specification. The equivalence is based on mapping the instances of the variables in the program into respective variables in the specification. The source program and the object specification are then equivalent in the sense that the respective mapped variables have the same value. The specification can be viewed as an abstract set of mathematical equations. It can also be viewed as a computational model. The computation finds values for all the variables which make all equations and declarations true. This computation may be envisaged in a most direct way as conducted by a hypothetical dataflow computer with a very large memory and number of processors which can execute the specification directly. The equivalence can be considered in terms of either view of the specification. The computational view is briefly reviewed in section 2.5.

We will not be concerned whether the source program "makes sense" or whether it is "correct", only that the object specification variables have the same values as those in the source program, respectively. In this sense, the algorithm in the object specification will be the same as that of the source program.

The translation will consist of a number of transformations. Starting with the source program, each transformation modifies a program into an equivalent program which is progressively closer to the object equational specification. Basically, the difference between a procedural program and an equational specification is as follows. Variables in a procedural program can have assigned to them none, one or several values in the course of sequential execution. In a mathematical equational specification, each variable can assume one and only one value. The transformations then rename instances of program variables and declare arrays so that each elemental variable has one and only one assigned value.

There will also be simplification transformations that will make the equational specification easier to understand and modify. Thus the objective is not only that the object specification is equivalent to the source program, but also that it be readable and understandable while exposing to the user to details of the inherent logic of the source program.

2.2 The Procedural Language

The subsequent discussion focuses on translation of a source program that contains only a selected subset of types of statements, shown in Table 1, into an equational specification. The reasons for using a subset of the types of statements are as follows. This subset of types of statements is common to practically all procedural languages - with differences only in syntax. The translation of a source program in different languages into an object equational specification is envisaged as consisting first of pre-translation of these programs into the basic types of statements. Thus there will be one pre-translation for each language into basic types of statement, and a single translation to an equational specification language. There may be then post-translations into respective other nonprocedural languages. Nearly all of the pre and post translations can be performed automatically. Translating subroutine calls, gotos and dynamic memory allocations into the basic types of statement presents special problems, as discussed below.

Subroutines (or subprocedures) are considered independent entities which are separately and independently translated into respective equational specifications. We adapt essentially the so called "object oriented approach". Namely, in the calling program, a subroutine is viewed as an opreation on its arguments. The subroutine call (in the calling program) is translated into an assignment statement. The left hand side of the assignment statement consists of a structured variable that contains the subroutine's output arguments. The right hand side consists of an expression, where the subroutine appears as the operation on the input arguments. It is necessary to identify the subroutine's input and output arguments, respectively. In an object oriented procedural language the input and output arguments are explicitly identified. In a language such as FORTRAN, that has COMMON declarations, it may be necessary to search many programs and subroutines to identify the input and output arguments. Therefore human assistance may be helpful in performing this task.

Goto statements are to be pre-translated into while statements. The algorithm for this pre-translation is given in [24]. The result of this pre-translation preserves the topology and has the same order of efficiency. It may however, use renaming of the same variables.

Finally, declarations and references of dynamically allocated variables are to be translated into static declarations and references. This will cause use of much more memory. This will not matter as the ultimate object equational specification uses unlimited amount of memory. The forward translation re-optimizes the memory usage.

Consider the entire source program as a tree. There is a node for each statement (for the statement types in Table 1). The types of nodes are:

(a) A root node representing the entire program.

(b) intermediate nodes for block statements: do,while, if then and if else.

(c) leaf nodes for input/output and assignment statements.

Edges emanate from the root and intermediate nodes to the nodes of their constituents. The constituents are ordered depth first, left to right, in the order of the statements of the program. Declaration statements can be inserted in the program tree as leaf nodes anywhere prior to their usage. These variables are always considered as local to the program where they are declared.

Each statement node is further parsed to reflect the statement structure.

Thus, a source program is initially parsed into the tree structure. Sucessive transformations modify names in the tree, add conditions and subscripts, and even add or delete entire nodes. This leads to the specification after the final transformation.

Leaf statements:

1. Input/output (read,write)

2. Assignment

Block statements:

3. Do

4. While

5. If <condition> then <block_1>

   else <block_2>

Declaration statements

6. Declaration of variables

Note: Goto replaced by while

Subroutine call replaced by assignments

Dynamic memory replaced by static memory

Table 1: Basic Types of Statements Used in the Source Program

2.3 The Equational Specification Language

This section describes briefly the syntax and semantics of the basic part of the MODEL equational specification language. The equational specifications produced by the translation are accepted by the present MODEL system[14] that generates programs in PL/1, C and Ada. The MODEL system has a "specification extension" phase where it tolerates omissions in the user provided specification and fills-in missing parts automatically. We will take advantage of this feature to simplify the translation.

The order of specification statements has no significance and they may be in an arbitrary order.

The core of the language are the *equation* statements used to express rules. They are summarized in Table 2. They have the syntax and meaning of regular algebra or boolean algebra (depending on the operators). A mix of the two algebras is obtained through use of the *if* operation (IF <boolean condition> THEN <expression_1> ELSE <expression_2>). The expressions may be in regular algebra; expression_2 is optional.

The left hand side of an equation statement contains only a variable name. This is the independent variable of the equation. The right hand side is an expression. Expressions consist of operations and variables.

Variables referenced in equations may be scalars or arrays. An array variable name must be followed by a subscript expression for each of its dimensions, in parenthesis. (A variable may also denote an entire tree structure of more elemental variables. This is not in the basic part of MODEL).

A specification must include a definition of the *size* of each different dimension of a variable. The definition is given through a declaration or through an equation. In the latter case, the size of a dimension variable is denoted by a *control variable*, which is used as a left hand side of a defining equation.

There are two ways of denoting a control variable: statically - by use of the prefix SIZE to denote the number of elements in a dimension, and dynamically - by use of the prefix END to denote a boolean vector where each element denotes whether the respective element of a variable is the last one in the dimension.

The equation applies (is true) while the subscripts assume any integer value in the range of 1 to the size of the respective dimension. Thus an equation as well as a variable may be a multidimensional array. The equation and the respective variables are null if the size of any dimension is zero.

Finally, there are several types of operations that can be used in expressions. The regular algebra arithmetic operations consist of =,+, *,/ and **. (There are presently no differential and integral operators). Logical operations consist of comparison operations, and, or and not. String operations consist of concatenation, search of a string and string replacement. The if-then-else operation has the three operands as shown above. There are many built-in functions, for common mathematical and data processing operations. There are also more specialized user defined functions. These functions have strict requirements on number and structure of operands.

The use of declaration statements is shown in Table 3. Note that only declarations of input/output variables are mandatory. The entire input and output must be declared (differently from procedural languages, where only the structure transferred in an input/output operation is declared). The syntax for describing a structure is briefly shown in Table 3. The declaration can be viewed as a multi-level tree. Levels are numbered and nested. Each node is given a name and a definition of its number of repetitions. The number of repetitions is also the size of the respective dimension. The root node may have a device type specification (further described in section 4). Each leaf node must have a primitive data type.

Interim variables can be declared similarly. This declaration is optional. If omitted, it is generated automatically in the extension phase of the MODEL system. The use of declarations will be illustrated later.

Finally there is a need for statements that define the specification name, its inputs(called SOURCE) and outputs (called TARGET). These are shown in Table 4. Note that there are three types of statements: A MODULE denotes a main program. It is not called therefore it is not necessary to show its arguments. A FUNCTION has only input arguments and a PROCEDURE has input, output and update arguments.

EQUATIONS:
i.e. < variable name > (< subscript expression >,...) = < expression >;

< variable name > may refer to an individual element variable or to a tree structure of subvariables.

Control variables denote the size of dimensions of arrays. They must be defined by equations. They can be represented as:

SIZE. < variable name > (<subscript expression>,...): denotes the number of elements in the rightmost dimension of <variable name >.

END. < variable name > (<subscript expression>,...): denotes whether an element is the last one in the rightmost dimension of <variable name>.

Subscript denotes the index of the referenced element of an array. The equation is true for all the integer values of each subscript in the range of 1 to the size of the respective dimension (defined by a constant or control variable). The equation and the array do not apply (are nullified) if the size of a dimension is zero. The syntax of subscripts is: sub1, sub2,... . These subscripts are local to the equation where they are used.

Operations used in expressions include the following:

arithmetic, logical and string operations

if-then-else

functions, built-in or defined in subroutines

Note that the translator from MODEL to procedural languages tolerate omissions of definitions of control variables.

Table 2: Equations, Variables, Subscripts and Operations in MODEL

DECLARATIONS:

Input/Output: Declaration of input/output structure is mandatory. The entire input/output data must be declared as a structure, down to individual data elements and their data types.

1 <structure name><repetitions><type of device,>

2 <substructure name><repetitions> ...,

. .

n <elementary variable name>... <repetitions><primitive data type>;

<repetition> may consist of:

<integer>

<min integer>:<max integer>, or

* : denoting 1 or more repetitions

The type of device must be declared as follows:

sequential (default) file,

messages from other processes (tasks),

addressed messages,

random access and shared memory.

There are no input/output commands.

Interim variables :Declaration is optional. The translator from MODEL to a procedural language declares interim variables automatically.

Table 3: Declarations of Input/Output and Interim Variables in MODEL

HEADER:

<u>Name Statements</u>

MODULE:<main procedure name>

FUNCTION:<function name> (<input_argument_1>,...);

PROCEDURE:<subroutine name> (<argument1>,...);

Input/Output Argument Declarations

SOURCE: <input argument_1>,...;

TARGET: <output argument_1>,...;

update arguments are both input and output

Table 4: Header Statements in MODEL

2.4 Transformations

Two series of transformations are discussed in this section. The first series, shown in Figure 2, transforms the source program into a form close to that of equations. The second set, shown in Figure 3, simplifies the initial equational specification into a form that is easier to read and understand. This section introduces the respective transformations. They are then illustrated in section 3 and further discussed in section 4.

The first set of transformations (Figure 2) transforms a source program into an equivalent program where variables are assigned a value once and only once. Each value assigned to a variable in the source program has a distinct variable in the object program. Two methods are used to define distinct variables in the object program. First, variables on the left hand side of different assignment statements in the source program are given different names in the object program. Next a variable that is assigned multiple values in a single assignment in a loop is transformed into an array in the object program, with one element for each iteration of the loop.

Additional variables are used to denote the index of elements in an array and the size of every dimension.

The same order of time efficiency is retained in the source and object programs of each transformation. There are the same order of number of assignments in both programs. There are cases where an IF condition selects either a THEN or an ELSE assignment to a variable (but not both). In cases that the IF condition does not select an assignment, then there is no need for a corresponding distinct element in the object program. (As will be discussed in Section 4, a specification can be simplified by deviating from this approach.)

The first pre-translation transformation in Figure 2 (the 0 transformation), translates the source program into an equivalent program using a subset of the types of statements. It is not further discussed in this report.

The first transformation collects THEN and ELSE assignments to the same variable. Wherever possible, it merges them into a single statement. (The dependencies in these statements on other variables must be checked to assure that they can be executed in the same place in a program).

The second transformation renames each variable if it is used in different assignment statements. It produces a renaming table showing (a) the renamed variables, (b) respective statements, (c) conditions where each renamed variable is used and (d) the subscripts used to index instances of assignments of the variable.

This is followed in the third transformation by generating an equivalent single-assignment program using the renamed variable.

The next translation (the fourth) declares memory space for an array for the distinct values assigned for the same variable in loop iterations. In this way the objective of one and only one value assigned to a variable is attained. This form of the program can be readily represented by equations.

The transformations in Figure 3 have the objective of producing a specification and simplifying it. The fifth transformation essentially copies the assignments produced previously as a set of equations. It then also generates declaration and heading statements.

The specification is now more complex than the source program, because a number of variables and conditions were added to make explicit all the interactions among variables. Transformations 7 and 8 simplify the specification in these respects. In the seventh transformation some variables are eliminated by substituting their defining expresssion in other equations. Further analysis is conducted in the eighth transformation to simplify conditions in equations. Conditions may be collected and factored and duplicates eliminated. Conditions may also be simplified when they do not depend on input data.

```
                 Source Procedural Program
                            |
                            V
+---+-----------------------------------------+
| 0 |  Pre-Translation                        |
|   |  Translate into basic types of statements |
+---+-----------------------------------------+
                            |
                            V
       Source program using only basic types of statements
                            |
                            V
+---+-------------------------------------------+
| 1 |Consolidate if-then with if-else assignments |
|   |                                           |
+---+-------------------------------------------+
                            |
                            V
              program with then-else statements
                            |
                            V
+---+-------------------------------------------+
| 2 |     Generate renaming table for           |
|   |         single assignments                |
+---+-------------------------------------------+
                            |
                            V
                      Renaming Table
                            |
                            V
+---+------------------------------------------+
| 3 |     Transform into single assignment     |
|   |                                          |
+---+------------------------------------------+
                            |
                            V
              program with only single assignments
                            |
                            V
+---+------------------------------------------+
| 4 |  Transform into single-value variables   |
|   |                                          |
+---+------------------------------------------+
                            |
                            V
              Program with single value variables
```

Figure 2: Program Transformations

```
        Program with single valued variable
                        |
                        V
+---+---------------------------------------+
| 5 |  Transformation into specification    |
|   |                                       |
+---+---------------------------------------+
                        |
                        V
              Initial specification
                        |
                        V
+---+---------------------------------------+
| 6 |  Transformation to reduce             |
|   |  variables and equations              |
+---+---------------------------------------+
                        |
                        V
          Reduced equations specification
                        |
                        V
+---+---------------------------------------+
| 7 |  Transformation to reduce ifs         |
|   |                                       |
+---+---------------------------------------+
                        |
                        V
         Simplified logic in specification
```

Figure 3: Specification Transformations

2.5 A Computational View of an Equational Specification

An equational specification can be viewed as a set of true declarations, equations and headings. A computational view of the specification may be considered directly in terms of a dataflow machine, with a processor for each array variable and for each equation, and a communication link for each dependency. The dataflow graph for such a machine is in fact generated by the MODEL system. It is called an *array-graph* [26,35]. It consists of a node for each variable and for each equation, and an edge for each dependency relation between a variable and an equation, and vice versa. The edge communicates the value of the variable and its indices in an array. Figure 4 illustrates this concept for a very simple example. Further discussion is provided in section 3.

A very simple specification is shown at the top of Figure 4. The corresponding computational view of the specification is shown at the bottom of the figure.

The specification, called *summer*, adds up the elements of an input *vector* and outputs the *sum_of_elements.* The specification starts with three header statements showing the name of the specifications, its source variables and target variables. They are followed by declaration of the source vector structure: 100 *lines*, each contains an *element*, and the sum-of-elements target structure: of one *total-line* containing the *total.* This is followed by one equation that uses the SUM function.

The corresponding dataflow machine is shown at the bottom of Figure 4. The processors for each variable are denoted by circles. The processor for the equation is denoted by a rectangle. Their functions of each processor are shown next to the respective node.

```
MODULE: summer;
SOURCE: vector;
TARGET: sum_of_elements;

1 vector is FILE
  2 line(100) is RECORD,
    3 element is FIELD(INTEGER);

1 sum_of_element is FILE,
  2 total_line is RECORD,
    3 total is FIELD(INTEGER);

total  = SUM(element(sub1),sub1);

                 vector-accesses a file

          |                  +----
          V                  |Reads and stores 100 line, sends each line
             line(100)--------|and its subscript as they are received.
                             +----
          |sub1
          V
              element --------+----
                             |Stores and distributes 100 elements and
          |sub1              |their subscripts as they are received.
          V
     +-----------------------------------+  +----
     |  total = sum(element(sub1),sub1)  |--| Adds when it receives
     +-----------------------------------+  | an element and its
          |                                 | subscript.  Sends the
          V                                 | total when all additions
                                            | have been completed.
             total                          +----
          |
          V

             total_line - prints total-line
          |
          V

             sum_of_elements - closes file
```

Figure 4: Example of a Specification and its Computational View

3. DESCRIPTION OF THE TRANSLATION ALGORITHM THROUGH AN EXAMPLE

3.1 The Example

The example selected to illustrate the transformations is shown in Figure 5. It is a FORTRAN program for computing the greatest common divisor (gcd) of two input integers (x and y). It has been selected because it illustrates well most of the transformations. The GCD example in Figure 5 is well known[27], short and very simple. Therefore it can be easily used within the bounds of this report. It utilizes only the basic types of statements (Table 1). Therefore the pre-translation (0 th transformation in Figure 2) can be omitted. It does not include an instance where the THEN and ELSE assignments to the same variable need to be merged into one statement. Therefore transformation 1 (Figure 2) can be skipped as well (it was discussed in section 2.4). While the example is short, it is also complex, as it handles implicitly the cases of:

gcd = input value of x = input value of y

gcd = input value of x

gcd = input value of y

by having a portion of the program skipped for each of these cases. The object specification will be shown to make these cases explicit.

```
c     PROGRAM GCD
c     FORTRAN example to find the greatest common divisor of
c     two positive integers.

      INTEGER x, y

      READ(5,100) x, y
  100 FORMAT(i4,i4)

      DO WHILE (x ^= y)
        IF x > y THEN
           x = x-y
        ELSE
           y = y-x
        ENDIF
      ENDDO

      gcd = x
      WRITE(6,200) gcd
  200 FORMAT(i4)
      END
```

Figure 5: Example of FORTRAN Program to Find the Greatest Common Divisor

There are two options for the transformation algorithms. The object specification can adhere closely to the source program algorithm and make the "side effects" explicit. This will lead to a more complex specification as it must show all the conditions explicitly. This option is useful for analyzing the operation of the source program. The other option is to avoid the "side effects" by changing the algorithm of the source program. This will lead to a simpler specification. The first option is employed below. The second option is discussed in section 4.

3.2 Renaming - Second Transformation

Assume that the source program has been parsed into a tree as discussed in section 2. The renaming is performed in the tree. There are three parts to the renaming.

Renaming LHS variables: In the first part the variable on the left hand side (lhs) of each equation is renamed. The new name is a concatenation of the variable name and the statement number (e.g. the variable x on the left hand side of statement 12 is renamed x_12). In this way there are distinct variables in each statement and the single assignment rule is applied to the program.

Computing indices of instances of assignments in loops: Figure 6 shows the transformation of WHILE loops. As shown, a WHILE loop is translated into two nested loops - DO and WHILE, respectively. The case when the source WHILE loop is skipped (i.e. sizek=0) is expressed by a DO loop, and the case when there are one or more repetitions (sizek=1) is expressed by the WHILE loop. In this way these two cases are differentiated explicitly. Sub1, sub2, etc., are the subscript values of the instances of assignment in the WHILE loop.

Source Program

```
WHILE <condition>

    <block>

END DO
```

Single Assignment Program

```
sizek = IF <initial condition>
        THEN 1 ELSE 0
DO sub1=1 to sizek
   sub2 = 0
   WHILE (IF sub2=0 THEN
          true ELSE ^endk) DO
      sub2 = sub2 + 1
      <block>
      endk = ^<condition>
   END DO
END DO
```

k is the statement number of the WHILE statement

sub2 serves as the counter for assignments in <block>

Figure 6: Transforming "WHILE"

The case that a block is nested in an if statement is illustrated in Figure 7. (Several other ways to handle this are described in section 4.) It is necessary to have a counter for the number of instances of assignments in an if block. The index of each assignment is a concatenation of "sl" and the statement number (e.g. sl11). *sl* is an abbreviation for *sublinear index*, namely an index which is a function of the subscript subk of the loop in which the assignment statement is nested. sl increases in steps of one or zero for each increase of one in subk. It always has an integer value (e.g. sl(subk) <=subk).

For each variable on the lhs of an assignment, the source program tree is scanned to determine the do, while and if blocks within which an assignment is nested. Statements are inserted for evaluating the respective subscripts or sublinears. The condition for terminating a while loop is named *endk*, where k is the number of the respective while statement.

Renaming RHS variables: In this part, the right hand side (rhs) variables of assignments are defined in terms of lhs variables, their subscripts, sublinear variables and the end variables. The tree is scanned to find for each rhs variable the same named lhs variables which has been assigned values preceding the rhs reference. There are a variety of cases where there are different preceding assignments. Each case will be a function of the subscripts of the nesting whiles, sublinears of the nesting ifs and of the end variables.

The renaming is illustrated in Figure 8 for the GCD example. It shows in the first column the source program variables: (x, y, and gcd). This is followed by the size and end variables: (size10 and end10), followed by the sublinears: (sl11 and sl13). For each variable, the table gives its statement numbers, whether the variables are on the rhs or lhs, their renaming and the respective subscripts of instances of lhs variables.

Source Program	**Single Assignment Program**
	slk = function (<condition>)
IF <condition>	IF<condition>
THEN	THEN
<block>	<block>
END IF	END IF

Figure 7: Transforming "IF<condition> THEN <block>"

var	statement address	position	renamed as	subscript
x	07	Read	x_7	scalar
	10b1	R	x_7	
	11b2	R	x_7 : [sub2=1]	
			IF sl11=0 THEN x_7 ELSE x_12	
	11b4	R	x_7 : [sub2=1]	
			IF sl11=0 THEN x_7 ELSE x_12	
	12	L	x_12	sub1,sl11(sub1,sub2)
	12	R	IF sl11=1 THEN x_7 ELSE x_12	
	14	R	IF sl11=0 THEN x_7 ELSE x_12	
	16b1	R	IF sl11=0 THEN x_7 ELSE x_12	
	18	R	IF size10=0 THEN x_7	
			ELSE IF sl11=0 THEN x_7	
			ELSE x_12	
y	07	Read	y_7	scalar
	10b1	R	y_7	
	11b2	R	y_7 : [sub2=1]	
			IF sl13=0 THEN y_7 ELSE y_14	
	11b4	R	y_7 : [sub2=1]	
			IF sl13=0 THEN y_7 ELSE y_14	
	12	R	IF sl13=0 THEN y_7 ELSE y_14	
	14	L	y_14	sub1,sl13(sub1,sub2)
	14	R	IF sl13=1 THEN y_7 ELSE y_14	
	16b1	R	IF sl13=0 THEN y_7 ELSE y_14	
gcd	18	L	gcd_18	scalar
	19	Write	gcd_18	
size10	10b1	L	size10	scalar
	10b1	R	size10	
	18	R	size10	
end10	10	R	end10	
	16b1	L	end10	scalar

Figure 8: Second Transformation - Renaming Table

sl11	11b1	R	sl11	
	11b2	L	sl11	scalar
	11b2	R	sl11	
	11b4	R	sl11	
	12	R	sl11	
	14	R	sl11	
	16b1	R	sl11	
	18	R	sl11	

sl13	11b2	R	sl13	
	11b3	R	sl13	
	11b4	L	sl13	scalar
	11b4	R	sl13	
	12	R	sl13	
	14	R	sl13	
	16b1	R	sl13	

Figure 8: Second Transformation - Renaming Table (Continued)

The read and write statements are treated same as assignment statements. However if a variable is read or written only once, the statement may be disregarded altogether as the object equational language has no input/output commands (e.g. if a program contains: ...write (5,x)...write (6,x)..., then this is equivalent to two assignment statements).

The renaming uses pseudo statements which are not directly executable in FORTRAN. The IF-THEN-ELSE operation can be nested in parenthesis like other arithmetic operations and functions. For example:

IF((IF cond THEN a ELSE b) > (IF cond THEN c ELSE d)) THEN...

is the same as

IF(cond & a>c)|(∧cond & b>d) THEN

This also allows the use of ifs on the rhs. For example

x = IF cond THEN a ELSE b

3.3 Single-Assignment Program - Third Transformation

Figure 9 shows the transformation of the source program through inserting in it the renamings shown in Figure 8. Mapped instances of variables have the same values. In this way the source and object programs are equivalent.

Note that the special cases in the source program are shown explicitly.

if gcd = x_7 and x_7 = y_7 then size 10=0 and x_12, x_14, sl11, sl13 are null.

if gcd = x_7 and y_7 ∧= x_7 then sl11(sub2)=0 always and x_12 is null.

if gcd = y_7 and x_7 ∧= y_7 then sl13(sub2)=0 always and y_14 is null.

```
01 c      PROGRAM GCD
02 c      FORTRAN example to find the greatest common divisor of
03 c      two positive integers.
04
05        INTEGER x_7, y_7, x_12, y_14, size10, sl11, sl13
05a1      LOGICAL end10
06
07        READ(5,100) x_7, y_7
08    100 FORMAT(i4,i4)
09
10b1      size10 = IF (x_7 ^= y_7) THEN 1 ELSE 0
10b2      DO sub1=1 to size10
10b3        sub2 = 0
10          DO WHILE ^(IF sub2=0 THEN false ELSE end10)
10a1          sub2 = sub2 + 1
11b1          sl11b = IF sub2=1 THEN 0 ELSE sl11
11b2          sl11 = IF sub2=1
                     THEN IF x_7 > y_7
                          THEN 1 ELSE 0
                     ELSE IF (IF sl11=0 THEN x_7 ELSE x_12)
                           > (IF sl13=0 THEN y_7 ELSE y_14)
                          THEN sl11 + 1 ELSE sl11
11b3          sl13b = IF sub2=1 THEN 0 ELSE sl13
11b4          sl13 = IF sub2=1
                     THEN IF ^(x_7 > y_7)
                          THEN 1 ELSE 0
                     ELSE IF ^((IF sl11=0 THEN x_7 ELSE x_12)
                             > (IF sl13=0 THEN y_7 ELSE y_14))
                          THEN sl13 + 1 ELSE sl13
12            x_12 = IF sl11 > sl11b
                     THEN (IF sl11=1 THEN x_7 ELSE x_12)
                        - (IF sl13=0 THEN y_7 ELSE y_14)
14            y_14 = IF sl13 > sl13b
                     THEN (IF sl13=1 THEN y_7 ELSE y_14)
                        - (IF sl11=0 THEN x_7 ELSE x_12)
16b1          end10 = ^((IF sl11=0 THEN x_7 ELSE x_12)
                     ^= (IF sl13=0 THEN y_7 ELSE y_14))
16b2          x_16 = IF end10 THEN (IF sl11=0 THEN x_7 ELSE x_12)
16          ENDDO
16a1      ENDDO
17
18        gcd_18 = IF size10=0 THEN x_7 ELSE x_16
19        WRITE(6,200) gcd_18
20    200 FORMAT(i4)
21        END
```

Figure 9: Third Transformation - Single Assignment Program

3.4 Single-Value Variables Program - Fourth Transformation

Figure 10 shows the GCD programs with single value assignments only. This is achieved by declaring an array for each variable assignment in a loop in Figure 9. If the assignment is nested in multiple loops then the array will be multi-dimensional. Note that the size of a dimension of an array may be a function of subscripts of higher order dimensions (of an outer loop).

Referencing a variable in a loop on the rhs of an assignment may require use of the subscript expression to point to a lower number element, e.g. sub2-1. Note that x_12 and y_12 are subscripted with sublinear variables sl11 and sl13 respectively.

This completes the transformations on programs. The assignments in Figure 10 may be read directly as equations.

```
01 c     PROGRAM GCD
02 c     FORTRAN example to find the greatest common divisor of
03 c     two positive integers.
04
05       INTEGER x_7, y_7, x_12(1,*), y_14(1,*)
05a1     INTEGER size10, sl11(1,*), sl13(1,*)
05a2     LOGICAL end10(1,*)
06
07       READ(5,100) x_7, y_7
08   100 FORMAT(i4,i4)
09
10b1     size10 = IF (x_7 ^= y_7) THEN 1 ELSE 0
10b2     DO sub1=1 to size10
10b3       sub2 = 0
10         DO WHILE ^(IF sub2=0 THEN false ELSE end10(sub1,sub2))
10a1         sub2 = sub2 + 1
11b2         sl11(sub1,sub2) = IF sub2=1
                               THEN IF x_7 > y_7 THEN 1 ELSE 0
                               ELSE IF (IF sl11(sub1,sub2-1)=0 THEN x_7
                                        ELSE x_12(sub1,sl11(sub1,sub2-1))
                                     > (IF sl13(sub1,sub2-1)=0 THEN y_7
                                        ELSE y_14(sub1,sl13(sub1,sub2-1))
                                    THEN sl11(sub1,sub2-1) + 1
                                    ELSE sl11(sub1,sub2-1)
11b4         sl13(sub1,sub2) = IF sub2=1
                               THEN IF ^(x_7 > y_7) THEN 1 ELSE 0
                               ELSE IF ^((IF sl11(sub1,sub2-1)=0 THEN x_7
                                          ELSE x_12(sub1,sl11(sub1,sub2-1))
                                       > (IF sl13(sub1,sub2-1)=0 THEN y_7
                                          ELSE y_14(sub1,sl13(sub1,sub2-1)))
                               THEN sl13(sub1,sub2-1) + 1
                               ELSE sl13(sub1,sub2-1)
12           x_12(sub1,sl11(sub1,sub2))
                  = IF sub2=1 & sl11(sub1,sub2)=1
                     | sub2>1 & sl11(sub1,sub2)>sl11(sub1,sub2-1)
                    THEN (IF sl11(sub1,sub2) = 1 THEN x_7
                            ELSE x_12(sub1,sl11(sub1,sub2-1)))
                       - (IF sl13(sub1,sub2) = 0 THEN y_7
                            ELSE y_14(sub1,sl13(sub1,sub2-1)))
14           y_14(sub1,sl13(sub1,sub2))
                  = IF sub2=1 & sl13(sub1,sub2)=1
                     | sub2>1 & sl13(sub1,sub2)>sl13(sub1,sub2-1)
                    THEN (IF sl13(sub1,sub2) = 1 THEN y_7
                            ELSE y_14(sub1,sl13(sub1,sub2-1)))
                       - (IF sl11(sub1,sub2) = 0 THEN x_7
                            ELSE x_12(sub1,sl11(sub1,sub2-1)))
```

Figure 10: Fourth Transformation - Program with Single Value Assignment to Variables

```
16b1        end10(sub1,sub2) = ^((IF sl11(sub1,sub2)=0 THEN x_7
                                  ELSE x_12(sub1,sl11(sub1,sub2)))
                              ^= (IF sl13(sub1,sub2)=0 THEN y_7
                                  ELSE y_14(sub1,sl13(sub1,sub2))))
16b2        x_16 = IF end10(sub1,sub2)
                   THEN (IF sl11(sub1,sub2)=0 THEN x_7
                         ELSE x_12(sub1,sl11(sub1,sub2)))
16            ENDDO
16a1        ENDDO
17
18          gcd_18 = IF size10 = 0 THEN x_7 ELSE x_16
19          WRITE(6,200) gcd_18
20    200   FORMAT(i4)
21          END
```

Figure 10: Fourth Transformation - Program with Single Value Assignment to Variables (Continued)

3.5 The Initial Equations in the Specification - Fifth Transformation

Figure 11 shows the equations in the specification. They are derived directly from the assignments in Figure 10. Note that there are no input/output or loop control statements in an equational specification, and that the input/output statements (e.g. for x_7,y_7 and gcd) need not be transformed into equations.

```
10b1  size10 = IF (x_7 ^= y_7) THEN 1 ELSE 0

11b2  sl11(sub1,sub2) = IF sub2=1
                        THEN IF x_7 > y_7 THEN 1 ELSE 0
                        ELSE IF (IF sl11(sub1,sub2-1)=0 THEN x_7
                                 ELSE x_12(sub1,sl11(sub1,sub2-1))
                              > (IF sl13(sub1,sub2-1)=0 THEN y_7
                                 ELSE y_14(sub1,sl13(sub1,sub2-1))
                             THEN sl11(sub1,sub2-1) + 1
                             ELSE sl11(sub1,sub2-1)

11b4  sl13(sub1,sub2) = IF sub2=1
                        THEN IF ^(x_7 > y_7) THEN 1 ELSE 0
                        ELSE IF ^((IF sl11(sub1,sub2-1)=0 THEN x_7
                                   ELSE x_12(sub1,sl11(sub1,sub2-1))
                                > (IF sl13(sub1,sub2-1)=0 THEN y_7
                                   ELSE y_14(sub1,sl13(sub1,sub2-1)))
                             THEN sl13(sub1,sub2-1) + 1
                             ELSE sl13(sub1,sub2-1)

12    x_12(sub1,sl11(sub1,sub2))
           = IF sub2=1 & sl11(sub1,sub2)=1
              | sub2>1 & sl11(sub1,sub2)>sl11(sub1,sub2-1)
             THEN (IF sl11(sub1,sub2)=1 THEN x_7
                   ELSE x_12(sub1,sl11(sub1,sub2-1)))
                - (IF sl13(sub1,sub2) = 0 THEN y_7
                   ELSE y_14(sub1,sl13(sub1,sub2-1)))

14    y_14(sub1,sl13(sub1,sub2))
           = IF sub2=1 & sl13(sub1,sub2)=1
              | sub2>1 & sl13(sub1,sub2)>sl13(sub1,sub2-1)
             THEN (IF sl13(sub1,sub2) = 1 THEN y_7
                   ELSE y_14(sub1,sl13(sub1,sub2-1)))
                - (IF sl11(sub1,sub2) = 0 THEN x_7
                   ELSE x_12(sub1,sl11(sub1,sub2-1)))

16b1  end10(sub1,sub2) = ^((IF sl11(sub1,sub2)=0 THEN x_7
                            ELSE x_12(sub1,sl11(sub1,sub2)))
                        ^= (IF sl13(sub1,sub2)=0 THEN y_7
                            ELSE y_14(sub1,sl13(sub1,sub2))))

16b2  x_16 = IF end10(sub1,sub2)
             THEN (IF sl11(sub1,sub2)=0 THEN x_7
                   ELSE x_12(sub1,sl11(sub1,sub2)))

18    gcd_18 = IF size10 = 0 THEN x_7 ELSE x_16
```

Figure 11: Fifth Transformation - Equations

3.6 Simplifying the Specification - Sixth and Seventh Transformation

In the interest of brevity the joint results of the sixth and seventh transformations are shown in Figure 12.

Variable substitution : The sixth transformation has the main objective to reduce the number of variables and equations. This is performed by substituting for a variable on the rhs its defining expression in an equation that defines the variable (on the lhs). The equations and respective lhs variables which are candidates for reduction are chosen trying to avoid excessively increasing the complexity and understandability of the remaining equations. The selection of equations and variables to be eliminated is influenced by two considerations. Prime candidates for elimination are "copying" equations. Namely, those equations that have only IF-THEN-ELSE operations on the rhs. Next, the graph of dependencies and the associated cycles that involve the equation that is a candidate for elimination, must be analyzed. The dependency graph is shown in Figure 13 and discussed further in section 3.7. Subscript expressions are attributes of edges in the dependency graph. Figure 12 shows the elimination of the variable and equation for x_16. Other examples that have been analyzed showed greater simplification due to use of this transformation.

Analysis of conditions: The seventh transformation consists mainly of analysis of selected conditions. Conditions in equations can be simplified by factoring out like conditions and reducing the depth of the IF-THEN-ELSE operations. Figure 12 shows the elimination of nesting of ifs and the definition of a new variable *slc11* which is common in the equations that define sl11 and sl13. The latter have been simplified by use of a *sublinear* function.

Analysis of conditions may also lead to simplifying the specification. The equations that define sizes of dimensions (END and SIZE prefixes) are analyzed. This may allow elimination of entire dimensions of variables, if it is possible to prove that sizek=1 or endk(subk)=subk=1.

```
MODULE: GCD;
SOURCE: File5;
TARGET: File6;

1 File5 IS FILE,
  2 inr IS RECORD,
    3 (x_7, y_7) ARE FIELDS (PIC 'zzz9');

1 Temp IS FILE,
  2 Tempf (0:1) IS RECORD,
    3 (x_12, y_14) (*) ARE FIELDS (PIC 'zzz9');

1 File6 IS FILE,
  2 outw IS RECORD,
    3 gcd_18 IS FIELD (PIC 'zzz9');

SIZE.Tempf = IF (x_7 ^= y_7) THEN 1 ELSE 0;

slc11(sub1,sub2) = IF sub2=1 THEN x_7 > y_7 ELSE
           IF sl11(sub1,sub2-1)=0 THEN x_7>y_14(sub1,sl13(sub1,sub2-1)) ELSE
           IF sl13(sub1,sub2-1)=0 THEN x_12(sub1,sl11(sub1,sub2-1))>y_7 ELSE
              x_12(sub1,sl11(sub1,sub2-1))>y_14(sub1,sl13(sub1,sub2-1));

sl11(sub1,sub2) = sublinear(slc11(sub1,sub2),sl11(sub1,sub2-1),sub2);

sl13(sub1,sub2) = sublinear(^slc11(sub1,sub2),sl13(sub1,sub2-1),sub2);

x_12(sub1,sl11(sub1,sub2)) = IF slc11(sub1,sub2) THEN
               IF sub2=1 THEN x_7-y_7 ELSE
               IF sl11(sub1,sub2)=1 THEN x_7-y_14(sub1,sl13(sub1,sub2-1)) ELSE
               IF sl13(sub1,sub2)=0 THEN x_12(sub1,sl11(sub1,sub2-1))-y_7 ELSE
                  x_12(sub1,sl11(sub1,sub2-1))-y_14(sub1,sl13(sub1,sub2-1));

y_14(sub1,sl13(sub1,sub2)) = IF ^slc11(sub1,sub2) THEN
               IF sub2=1 THEN y_7-x_7 ELSE
               IF sl11(sub1,sub2)=0 THEN y_14(sub1,sl13(sub1,sub2-1))-x_7 ELSE
               IF sl13(sub1,sub2)=1 THEN y_7-x_12(sub1,sl11(sub1,sub2-1)) ELSE
                  y_14(sub1,sl13(sub1,sub2-1))-x_12(sub1,sl11(sub1,sub2-1));

END.sl11(sub1,sub2) = IF sl11(sub1,sub2)=0 THEN x_7=y_14(sub1,sl13(sub1,sub2))
         ELSE IF sl13(sub1,sub2)=0 THEN x_12(sub1,sl11(sub1,sub2))=y_7
           ELSE x_12(sub1,sl11(sub1,sub2))=y_14(sub1,sl13(sub1,sub2));

gcd_18 = IF SIZE.Tempf = 0 THEN x_7
              ELSE IF END.sl11(sub1,sub2) THEN IF sl11(sub1,sub2)=0 THEN x_7
                ELSE x_12(sub1,sl11(sub1,sub2));
```

Figure 12: Final Specification

3.7 The Array Graph

Figure 13 shows the array graph for the GCD specification in Figure 12. This graph is constructed automatically by the MODEL system. It shows:

variable nodes by circles,

equation nodes by rectangles,

dependencies by edges,

dimensionality is shown as attribute of each node,

subscript expression is shown as attribute of each edge

In the interest of clarity of the graph in Figure 13 the dependencies of nodes on sizes of respective dimensions are not shown.

The graph can be viewed as a dataflow machine computational model of the GCD program. Each node is a processor. Each edge is a communication link. Each node has four types of inputs: variable values, their subscripts, sizes of their dimensions (end and size array elements) and their subscripts. Each node has two types of outputs: variable values and their subscripts.

The nodes behave like Petrinet nodes. Whenever sufficient values and appropriate subscripts are input to a node, the respective output element, as specified by the respective equation or declaration, is immediately produced.

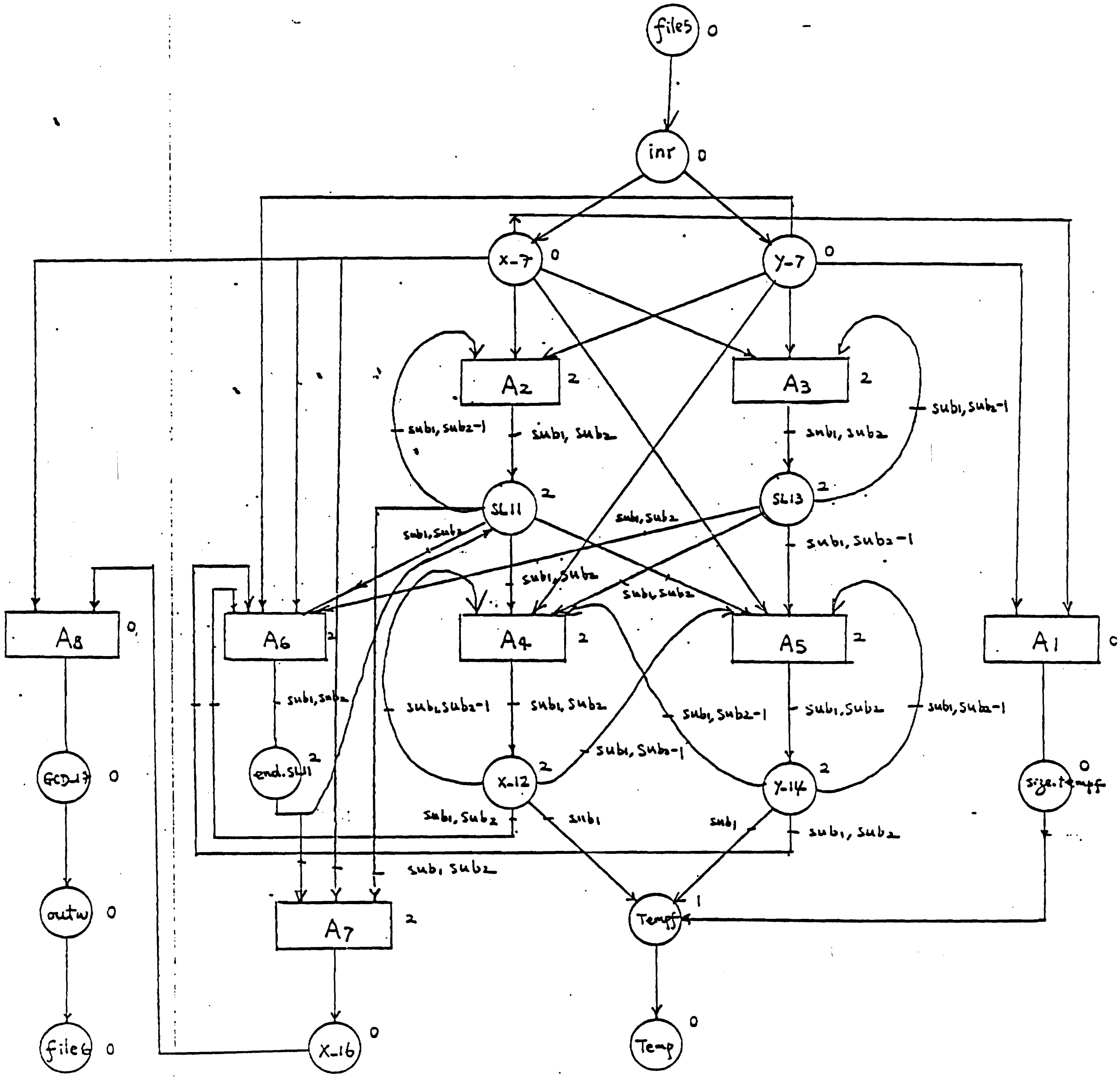

Figure 13: Array Graph of the GCD Dataflow Machine

4. DISCUSSION OF THE EQUATIONAL SPECIFICATION

As stated in section 1, among the objectives of the procedural to equational translation has been to provide a mathematical representation that is more explicit, readily understandable, suitable for manipulations needed in the verifications and, most important, easy to modify for program maintenance. This section touches on these issues.

The simplification transformations described in section 3.6 have the above objectives in mind. The simplification is illustrated by comparing Figures 11 and 12. The object equational specification in Figure 12 is still larger and more complex, than the source program of Figure 5 due to the use of subscripts and sublinears. Note that the simplification algorithm proved more effective in other examples that we investigated. On one hand, the object equational specification must provide more information about the algorithm that is used in the program, explicitly showing the side effects which are only implicit in the program. Therefore the specification will naturally be longer and more complex. On the other hand, in many cases (not illustrated by the above example) it is possible to use substitutions to eliminate implementation details, such as those involved in memory management, resulting in a simpler and more abstract equational specification.

The issues of simplification and use of the equational specification, for the variety of objectives listed above, obviously require additional research. This section explores two aspects. Section 4.1 explores using an alternate translation algorithm that produces a simplified less explicit equational specification. Section 4.2 explores use of an equational specification in verifying correctness.

4.1 Modifying the Translation to Produce a Simpler Specification

In some cases it is possible to eliminate the use of the sublinear subscripts (e.g. sl11 and sl13 in Figure 12), thus simplifying the resultant equational specification. This is shown below with the aid of the same example that was used in section 3 (Figure 5). As will be shown this simplification is not always possible and in some cases it does not materially simplify the resultant specification. Also the resultant specification does not follow the source program algorithm as closely as the one produced by the algorithm used in section 3. For different objectives (e.g. for understanding vs. verification), it may be preferable to use the different translation algorithms, (in this section vs. in section 3.2, respectively).

The difference in the algorithm is in translating the IF <condition> THEN <block> in Transformation 2. Instead of the translation shown in Figure 7, we employ the translation shown in Figure 14. (Note that instances where there is

```
IF <condition> THEN x=. . .
                ELSE x=. . .
```

have already been integrated into a single statement in Transformation 1.) The transformation in Figure 14 can be used only on assignments in the <block>; i.e. it cannot be used for input or output statements. Also if the <block> is large or it contains WHILE or DO statements, then the propogation of the IF <condition> to the nested assignments adds to the complexity of the resulting specification.

Source Program	**Single Assignment Program**
IF <condition>	
THEN	
x=<expression 1>	x = IF <condition> THEN <expression>
	ELSE x
y=<expression>	y = IF <condition> THEN <expression 2>
	ELSE y
. . .	. . .
ENDIF	

Figure 14: Alternative Transforming "IF <condition> THEN <block>"

The transformation in Figure 14 has been applied to the example in Figure 5. It yields the renaming table and program shown in Figures 15 and 16 respectively.

The final equational specification using the transformation in Figure 14 is shown in Figure 17. It is simpler than the one in Figure 12, due to the absence of the sublinear subscript sl13 and the sublinear condition slc11. The specification in Figure 12 distinguishes explicitly the cases

sl11(sub2)=0	gcd_18 = x_7
sl13(sub2)=0	gcd_18 = y_7

but these cases can not be distinguished in the specification of Figure 17.

This will be further discussed in section 4.2.

variable	statement address	position	renamed as	subscripts will be added
x	7	Read	x_7	scalar
	11	R	x_11	
	12	L	x_12	sub1,sub2
	12	R	IF sub2=1 THEN x_7 ELSE x_12	
	14	R	IF sub2=1 THEN x_7 ELSE x_12	
	18	R	IF size9=0 THEN x_7 ELSE x_16	
y	7	Read	y_7	scalar
	11	R	y_11	
	14	L	y_14	sub1,sub2
	14	R	IF sub2=1 THEN y_7 ELSE y_14	
	12	R	IF sub2=1 THEN y_7 ELSE y_14	
gcd	18	L	gcd_18	scalar
	18	Write	gcd_18	scalar
end10	10	R	end10	
	16b1	L	end10	
	16b2	R	end10	
size10	10b1	L	size10	scalar
	10b2	R	size10	
	18	R	size10	

Figure 15: Second Transformation - Renaming Table

```
1 C     PROGRAM GCD
2 C     Fortran example to find greatest common divisor of two
3 C     positive integers
4 C
5       INTEGER x_7, y_7, x_12, y_14, sub1, sub2, gcd_18
        LOGICAL end10
6
7       READ(5,100) x_7, y_7
8  100  FORMAT(i4,i4)
10b1    size10 = IF x_7=y_7 THEN 0 ELSE 1
10b2    DO sub1=1 to size10
10b3      sub2 = 0
10        WHILE (IF sub2=0 THEN true ELSE ^end10) DO
10a1        sub2 = sub2 + 1
11b1        x_11 = IF sub2=1 THEN x_7 ELSE x_12
11b2        y_11 = IF sub2=1 THEN y_7 ELSE y_14
12          x_12 = IF x_11>y_11 THEN (IF sub2=1 THEN x_7 ELSE x_12) -
                               (IF sub2=1 THEN y_7 ELSE y_14)
              ELSE IF sub2=1 THEN x_7 ELSE x_12;
14          y_14 = IF x_11>y_11 THEN (IF sub2=1 THEN y_7 ELSE y_14) -
                               (IF sub2=1 THEN x_7 ELSE x_12)
              ELSE IF sub2=1 THEN y_7 ELSE y_14

16b1        end10 = (x_12=y_14)
16b2        x_16 = IF end10 THEN x_12
16        END DO
        END DO
17
18      gcd_18 = IF size10=0 THEN x_7 ELSE x_16
19      WRITE(6,200) gcd_18
20 200  FORMAT(i4)
21      END
```

Figure 16: Third Transformation - Single Assignment Program

```
MODULE: GCD;
SOURCE: file5;
TARGET: file6;

   1 file5 is FILE,
     2 inr is RECORD,
       3 (x_7, y_7) are fields (pic 'zzz9');

   1 file6 is FILE,
     2 outr is RECORD,
       3 gcd_18 is FIELD(pic 'zzz9');

      x_12(sub1,sub2) = IF sub2=1 THEN IF x_7>y_7 THEN x_7-y_7 ELSE x_7
           ELSE IF x_12(sub1,sub2-1)>y_14(sub1,sub2-1) THEN
               x_12(sub1,sub2-1)-y_14(sub1,sub2-1) ELSE x_12(sub1,sub2-1);

      y_14(sub1,sub2) = IF sub2=1 THEN IF x_7>y_7 THEN y_7 ELSE y_7-x_7
           ELSE IF x_12(sub1,sub2-1)>y_14(sub1,sub2-1) THEN
               y_14(sub1,sub2-1) ELSE y_14(sub1,sub2-1)-x_12(sub1,sub2-1);

      end.x_12(sub1,sub2) = (x_12(sub1,sub2)=y_14(sub1,sub2));

      x_16 = IF end.x_12(sub1,sub2) THEN x_12(sub1,sub2);

      gcd_18 = IF size.tempf=0 THEN x_7 ELSE x_16;

      size.tempf = IF x_7=y_7 THEN 0 ELSE 1;

1 temp is FILE,
  2 tempf(0:1) is GROUP,
    3 (x_12, y_14)(*) is FIELD(pic 'zzzzzzz9');
--------------------------------------------------------------------
```

Figure 17: Final Specification

4.2 Proving Correctness of an Equational Specification

The correctness of the program in Figure 5 has been proven in [27]. This section shows a similar proof carried out on the equivalent equational specification in Figure 12. The main difference is that the proof in [27] requires drawing a graph of the program in Figure 5 and developing its path expressions. The proof based on the equational specification in Figure 12 does not require any graph analysis. It involves only analysis of conditions and substitution for variables defined in the lhs the respective rhs expressions of equations.

Three verification assertions, same as in [27], are shown in Figure 18. They show the behavior of a GCD function of two integer arguments (v,w). The value of this function is the greatest common divisor of the two arguments.

Figure 19 shows that the quational specification of Figure 12 preserves the assertions in Figure 18 by actually incorporating them under respective conditions. We prove that gcd_18 in Figure 12 is the value of the function GCD(V,W) where V,W may be x_7, y_7 respectively, or the respective sub2 elements of x_12(sl11(sub2)),y_14(sl13(sub2)) or some special shown mixes of them. (Note that sub2 is in effect a universal quantifier; an equation is true for all values of sub2 within the specified dimension size. Sub2 is a local subscript and may have a different range in a different equation).

1. v=w GCD(v,w) = v

2. v>w GCD (v,w) = GCD(v-w,w)

3. v<w GCD (v,w) = GCD(w-v,v)

Figure 18: Verification Assertions for the function GCD From [27]

Case 1: Preserving assertion 1 in Figure 18

subcase 1: x_7 = x_7

size10=0; slc11, x_12, y_14, end10 are null size.

gcd_18 = GCD(x_7,y_7) = x_7.

using equation for gcd_18.

subcase 2: x_7 = y_14(sl13(sub2)) = end10(sub2)

size10=1; sub1=1; sl11(sub2)=0; x_12 is null size.

gcd_18 = GCD(x_7,y_14(sl13(sub2))) = x_7.

using equations for gcd_18, x_16, slc11, and end10.

subcase 3: x_12(sl11(sub2)) = y_7 = end10(sub2)

size10=0; sub1=1; sl13(sub2)=0; y_14 is null size.

gcd_18 = GCD(x_12(sl11(sub2)),y_7) = x_12(sl11(sub2)).

using equations for gcd_18, x_16, slc11, and end10.

subcase 4: x_12(sl11(sub2)) = y_14(sl13(sub2)) = end10(sub2)

size10=1; sub1=1; sl11(sub2)>0; sl13(sub2)>0.

gcd_18 = GCD(x_12(sl11(sub2)),y_14(sl13(sub2))) = x_12(sl11(sub2)).

using equations for gcd_18, x_16, sl11, sl13, and end10.

Case 2: Preserving assertion 2 in Figure 18

subcase 1: x_7 > y_7

size10=1; sub1=1; sub2=1, sl13(sub2)=0; slc11(sub2)=true.

GCD(x_7,y_7) = GCD(x_12(sl11(sub2)),y_7) = GCD(x_7-y_7,y_7).

using equation for x_12.

Figure 19: Verification of Specification in Figure 12

subcase 2: x_12(sl11(sub2-1)) > y_7

size10=1; sub1=1; sub2>1; sl13(sub2)=0; slc11(sub2)=true.

GCD(x_12(sl11(sub2-1)),y_7) = GCD(x_12(sl11(sub2)),y_7)

= GCD(x_12(sl11(sub2-1))-y_7,y_7).

using equation for x_12.

subcase 3: x_7 > y_14(sl13(sub2-1))

size10=1; sub1=1; sub2>1; sl11(sub2)=0; slc11(sub2)=true.

GCD(x_7,y_14(sl13(sub2-1))) = GCD(x_7,y_14(sl13(sub2)))

= GCD(x_7-y_14(sl13(sub2-1)),y_14(sl13(sub2-1))).

using equation for y_14.

subcase 4: x_12(sl11(sub2-1)) > y_14(sl13(sub2-1))

size10=1; sub1=1; sub2>1; sl11(sub2)>0; sl13(sub2)>0; slc11(sub2)=true.

GCD(x_12(sl11(sub2-1)),y_14(sl13(sub2-1)))

= GCD(x_12(sl11(sub2)),y_14(sl13(sub2)))

= GCD(x_12(sl11(sub2-1))-y_14(sl13(sub2-1)),y_14(sl13(sub2-1))).

using equation for x_12.

Case 3: Preserving assertion 3 in Figure 18

symmetrical to case 2 except y > x.

Figure 19: Verification of Specification of Figure 12 (continued)

Figure 19 shows the cases in which there is conformance with the respective assertions. For each case there are subcases for the different conditions under which the assertion is preserved.

Case 1 shows that gcd_18 (see Figure 12) is equal to the GCD function for the four subcases when its two arguments are equal.

Case 2 and 3 show that the GCD function has the same value of gcd_18 for, not only for the arguments in case 1 where end10=true, but also for all the respectively shown arguments of x_7, y_7 and/or the same sub2 elements of x_12 and y_14.

Case 3 is not shown in detail as it is symmetrical with Case2.

The proof method consists of examining the conditions in the specification to make a classification of respective cases and subcases. We then use substitution to demonstrate systematic conformance with the assertions in each case and subcase.

5. CONCLUSION

We have posed in section 1 the problem of Reverse Software Engineering as "how to utilize outdated programs to reduce cost of developing new replacement systems." The emphasis is on *reducing cost* of *replacment systems.* The old systems are assumed to be inadequate in functionality and implementation technology. Still, to reduce cost it is desired to find and reuse what is available in the old system as a basis for making appropriate changes, deletions and additions.

Mathematical representations of programs have been widely claimed to be advantageous for understanding, checking and modifying software. Translation into a mathematical representation has been the constant theme in research into a number of directions concerning procedural programs. The underlying notion of this report is to use a mathematical representation as an intermediate step in Reverse Software Engineering. It is proposed as the medium for understanding, analyzing and changing old programs.

Many of the mathematical representations of procedural programs proposed in the past involved unfamiliar syntax and semantics. The choice here has been to use the widely known regular and boolean algebras as the syntax and semantics of the mathematical representation. The MODEL system is based on this syntax and semantics. It translates the equational specifications into procedural programs. We have extensive experience with the MODEL system in using equational specifications for software development. The objective of this research has been to investigate its effectiveness for Reverse Software Engineering.

There are then two questions to which we have sought answers:

1. What is the algorithm for translating a procedural program into an equational specification?

2. What is the relative effectiveness of using the result of the translation for understanding, analysis, proving correctness and maintaining programs?

The answer to the first question has been provided in sections 2 and 3. Once the underlying concepts are defined, the algorithm is straight-forward and can be implemented readily.

A definitive answer to the second question will require additional research. We have investigated many examples of procedural programs translated into equational specifications by the algorithm of section 3. Section 4.1 shows how two versions of equational specifications can be generated - one that is simpler and easier to understand, and one more complicated but which is more useful for analysis and verification. Section 4.2 shows by example the approach to verification based on equational specifications. We assume that if the proof is easier then true understanding is also easier.

A number of mathematical representations of procedural programs have been proposed. It is necessary to conduct a comparative study of their effectiveness vs. equational specifications, for the respective directions for which they have been proposed. This investigation will yield important insights into the usefulness of the different syntax and semantics of mathematical representations of procedural programs.

It is also necessary to conduct more extensive experimental research by automating the translation algorithm and processing larger and more complex program translations.

6. REFERENCES

1. T. Agerwala, Arvind, "Data Flow Systems," *Computer*, February, 1982.

2. J.R. Allen and K. Kennedy, "Automatic Loop Interchange," *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices V19 #6, June 1984.

3. E. Ashcroft, Z. Manna, "The Translation of Goto Programs to While Programs", *Proceedings, IFIP Congress 1971*, North-Holland Publ. Co. Amsterdam, pp. 250-255, 1972.

4. E. Ashcroft and W.W. Wadge, "Lucid, A Nonprocedural Language with Iteration," *Communications of the ACM*, V20 #7, July 1977.

5. J. Backus, "Can Programming be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, V21 #8, August, 1978.

6. B.S. Baker, "An Algorithm For Structuring Flowgraphs," *Journal of the ACM*, V24 #1, January 1977.

7. R. Balzer, "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering*, V7 #1, January 1981.

8. J. Baron, B. Szymanski, E. Lock and N. Prywes, "An Argument for Nonprocedural Languages," *Proc. Workshop Role of Languages in Problem Solving-1*, 1985.

9. S. Basu and J. Misra, "Proving Loop Programs," *IEEE Transactions on Software Engineering*, V1 #1, March 1975.

10. R.K. Boxer, "A Translator From Structured Fortran to Jovial/J73," *Proc. of the IEEE National Aerospace and Electronics Conference (NAECON-83)*, 1983.

11. J.M. Boyle and M.N. Muralidharan, "Program Reusability Through Program Transformation," *IEEE Transactions on Software Engineering*, V10 #5, September 1984.

12. E. Bush, "The Automatic Restructuring of Cobol," *Proc. of the IEEE Conf. on Software Maintenance*, November 1985.

13. T. Cheng, E. Lock and N. Prywes, "Use of Very High Level Languages and Program Generation by Management Professionals," *IEEE Transactions on Software Engineering*, V10 #5, September 1984.

14. Computer Command and Control Company, "The MODEL language Usage and Reference Guide – Non-Procedural Programming for Non-Programmers," 2401 Walnut, Philadelphia, PA 19103, 1987.

15. Digital Equipment Corporation, "VAX-11 Fortran User's Guide," *Software Distribution Center, Digital Equipment Corporation*, Maynard, MA 01754, 1979.

16. C.G. Faust, "Semiautomatic Translation of Cobol into Hibol," *(MS Thesis) MIT/LCS/TR-256*, March 1981.

17. R.W. Floyd, "Assigning Meaning to Programs," in *Proc. Symp. Applied Math.*, vol. 19, pp. 19-32, 1967.

18. M.J.C. Gordon, "The Denotational Description of Programming Languages, An Introduction," Springer-Verlag, 1979.

19. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, V12 #10, October 1969.

20. C.R. Hollander, "Decompilation of Object Programs," *Stanford Electronics Lab TR 54*, January 1973.

21. G.L. Hopwood, "Decompilation", *Ph.D. Thesis*, University of California, Irvine, 1978.

22. B.C. Housel III, "A Study of Decompiling Machine Languages Into High-Level Machine Independent Languages", *Ph.D. Thesis*, Purdue University, August 1973.

23. B.C. Housel and M.H. Halstead, "A Methodology for Machine Language Decompilation," *IBM Research Report RJ1316 (No. 20557)*, 17 pages, December 6, 1973.

24. S. Katz and Z. Manna, "Logical Analysis of Programs," *Communications of the ACM*, V19 #4, April 1976.

25. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symp.* pp.207-218, 1981.

26. K.S. Lu, "Program Optimization Based on a Non-Procedural specification," Ph.D. dissertation, Department of Computer Science, University of Pennsylvania, 1981.

27. Z. Manna, "Mathematical Theory of Computation," McGraw Hill Book Company, 1974.

28. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation By Machine," *Communications of the ACM*, V3 #4, April, 1960.

29. J.R. McGraw, "The VAL Language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, V4 #1, January, 1982.

30. M.J. O'Donnell, "Equational Logic as a Programming Language," The MIT Press, 1985.

31. H. Partsch, R. Steinbruggen, "Program Transformation Systems," *Computing Surveys*, V15 #3, September, 1983.

32. L. Paulson, "Compiler Generation from Denotational Semantics," in *Methods and Tools for Compiler Construction*, ed. by B. Lorho, pp. 219-250, 1984.

33. K.M. Pitman, "A Fortran to Lisp Translator," *Proc. of the 1979 Macsyma Users' Conference*, June 1979.

34. U. Pleban, "Compiler Prototyping Using Formal Semantics," SIGPLAN Notices, V19 #6, Montreal, June, 1984.

35. N. Prywes and A. Pnueli, "Compilation of Nonprocedural Specifications into Computer Programs," *IEEE Transactions on Software engineering*, V9 #3, May 1983.

36. C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman and C.E. Hewitt, "Programming Viewed as an Engineering Activity", MIT, Cambridge, MA, MIT/AIM-459, January 1978.

37. J. Samet, "Experience with Software Conversion," *Software – Practice and Experience*, V11 #10, 1981.

38. D. Schmidt, "Denotational Semantics - A Methodology for Language Development," Allyn and Bacon, Inc., 1986.

39. M. Shaw and W.A. Wulf, "Abstraction And Verification in ALPHARD: Defining and Specifying Iteration and Generators," *Communications of the ACM*, V20 #8, August 1977.

40. L. Sterling, E. Shapiro, "The Art of Prolog," *The MIT Press*, 1986.

41. B. Szymanski, E. Lock, A. Pnueli and N. Prywes, "On the Scope of Static Checking in Definitional Languages," *Proc. of the ACM Annual Conference*, San Francisco, CA, pp.197-207, October 1984.

42. B. Szymanski, N. Prywes, "Efficient Handling of Data Structures in Definitional Language," *Science of Computer Programming*, pp.221-245 No. 10, October 1988.

43. R.C. Waters, "A System for Understanding Mathematical FORTRAN Programs," MIT, Cambridge, MA, MIT/AIM-368, August 1976.

44. R.C. Waters, "Expressional Loops," *Proc. 10th ACM Symposium Principles of Programming Languages*, pp. 1-10, ACM, 1983.

45. R.C. Waters, "A Method for Analyzing Loop Programs," *IEEE Transactions on Software Engineering*, V11 #11, November 1985.

46. R.C. Waters, "Program Translation Via Abstraction and Reimplementation," *IEEE Transactions on Software Engineering*, V14 #8, August 1988.