December 1989

Reverse Software Engineering of Concurrent Real Time Programs

Mitchell C. Song
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-81.

Reverse Software Engineering of Concurrent Real Time Programs

Abstract

This paper presents an algorithm for translating concurrent procedural language programs into nonprocedural, mathematical language programs, called specifications. The goal is to achieve reuse of old existing programs in developing new systems, through having them explained automatically and facilitating their modification.

Mathematical languages are widely believed to be superior to procedural languages. Unlike procedural languages, mathematical languages do not have "side effects" and are oblivious to computer concepts. Thus mathematical languages free the user of having to "think like a computer" when developing or modifying a program. Its mathematical semantics make proving software correctness easier and improves software reliability. The specification can then be used to generate automatically highly efficient procedural language programs for computer system.

The translation algorithm centers around the difference in the meaning of variables in procedural and mathematical languages. In a procedural language a variable may be assigned many values. In a mathematical language, however, a variable may be assigned only one value. The translation algorithm focuses on renaming variables in a procedural language program so that each variable is assigned only one value.

This paper also presents a methodology for proving specification correctness. The idea is based on generating scenarios that define values of variable for an applicable situation and using this to prove the specification satisfy a given requirement. This is contrasted with use of temporal logic for proving correctness of concurrent programs.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-81.

Reverse Software Engineering Of Concurrent Real Time Programs

MS-CIS-89-81

Mitchell C. Song

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

December 1989

UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

REVERSE SOFTWARE ENGINEERING OF CONCURRENT REAL TIME PROGRAMS

Mitchell C. Song

Philadelphia, Pennsylvania
December 1989

A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Noah S. Prywes

Dr. Noah S. Prywes

Jean H. Gallier

Dr. Jean H. Gallier

ABSTRACT

This paper presents an algorithm for translating concurrent procedural language programs into nonprocedural, mathematical language programs, called specifications. The goal is to achieve reuse of old existing programs in developing new systems, through having them explained automatically and facilitating their modification.

Mathematical languages are widely believed to be superior to procedural languages. Unlike procedural languages, mathematical languages do not have "side effects" and are oblivious to computer concepts. Thus mathematical languages free the user of having to "think like a computer" when developing or modifying a program. Its mathematical semantics make proving software correctness easier and improves software reliability. The specification can then be used to generate automatically highly efficient procedural language programs for computer system.

The translation algorithm centers around the difference in the meaning of variables in procedural and mathematical languages. In a procedural language a variable may be assigned many values. In a mathematical language, however, a variable may be assigned only one value. The translation algorithm focuses on renaming variables in a procedural language program so that each variable is assigned only one value.

This paper also presents a methodology for proving specification correctness. The idea is based on generating scenarios that define values of variable for an applicable situation and using this to prove the specification satisfy a given requirement. This is contrasted with use of temporal logic for proving correctness of concurrent programs.

Contents

List of Figures

List of Tables

1 INTRODUCTION

This report deals with the reverse translation of concurrent programs using shared memory into equational specifications. It is an extension of the research performed on "Reverse Software Engineering" in [11]. The goal is to achieve reuse of existing software, thereby reducing the cost of redeveloping or modifying existing software to perform on different computer systems. The research for this report was supported by the Air Force Office of Scientific Research under contract AFOSR-88-0116.

1.1 The Problem

Programs written in procedural languages are computer dependent. They are prescriptive and consist of an ordered set of statements that direct the computer. Such existing programs must be frequently modified or redeveloped to take advantage of features offered by different computer systems. As the need to use modern computer systems increases, this costly approach is used widely. The difficulty and expense lie in the developer/modifier having to understand the old programs in order to modify and test them.

1.2 The Solution

Procedural language programs are notoriously difficult to read and comprehend. There are two major difficulties in working with procedural languages. The first is the need for the user to "think like a computer" in both developing and reading a program. A program is written as a sequence of steps taken by a computer. This is regarded by Parnas as the primary reason that conventional software development does not produce reliable programs [6, 10]. The second difficulty arises from the existence of "side effects". Side effects, though making a program more efficient, frequently obliterate from view the original algorithm on which the program is based. They add to the difficulties of understanding and modifying statements, as the statement may effect or be effected by other statements [11].

Mathematically oriented languages, on the other hand, are computer independent. They differ in that they are purely declarative or descriptive [11]. A specification contains only rules or axioms of the problem and is oblivious to computer concepts. There is no significance to the order of statements and no "side effects". Therefore, the user is freed from having to "think like a computer". They also state the algorithm explicitly. For these reasons, languages of this class are generally thought of as being easier to work with. The user only specifies what he/she wants and does not concern himself/herself with how to best utilize a given computer system. A forward translator can generate automatically an optimized program for a given computer system, taking advantage of any special features that system might offer.

We use proving correctness as a criterion for understanding a program. Equational languages are often easier to prove correct by using conventional algebraic manipulations. Since the program is nothing more then mathematical equations, the proof can be conducted mathematically. This is generally easier and leads to a more rigorous and straight forward proof. Thus, the use of equational languages can greatly reduce the cost of software development, maintenance and verification of correctness.

In light of the advantages of equational languages, it is very desirable to translate existing software into equational specifications before performing further maintenance or development. If all programming and maintenance is performed on a non-procedural, computer independent language, only one source version need to be kept. Programs can then be automatically generated in any language for any computer system using the non-procedural language program.

1.3 Contributions

We present in this report a reverse translation algorithm to translate concurrent programs using shared memory. An algorithm of this sort can greatly increase the life of existing software and reduce the cost of porting software to different computer systems. Porting

to another system merely consists of automatically forward translating the specification to the desired language on the desired computer system. As mentioned, multiple copies of the source for the same software are no longer needed. Maintenance is also made more cost effective. The specification is easier to comprehend and only one version need be modified.

As understanding of a program is the key objective, this document focuses on a methodology of using the generated specification for verification. This is also the objective of temporal logic, and the paper compares the use of specifications with the methodology of temporal logic.

1.4 Outline

Section 2 briefly describes Reverse Software Engineering and gives a summary of the reverse translation algorithm. Section 3 is provided for those not familiar with the MODEL equational language. It describes the basic components and their syntax. It also describes additional features such as the RENAMING clause that are used in the specifications of programs that share memory. Section 4 discusses the two forms of communication between concurrent programs, message passing and shared memory. It also explains how MODEL views shared memory. Section 5 gives in detail the algorithm for reverse translating concurrent programs that use shared memory. Section 6 presents a technique for proving specification correctness. Section 7 surveys temporal logic and contrasts the its proof methodology with that in Section 6. Finally, Section 8 summarizes the key issues in this paper.

2 SUMMARY OF REVERSE SOFTWARE ENGINEERING

Reverse Software Engineering is a process by which a procedural program is transformed, through a series of transformations, into an equational specification. This specification is the mathematical meaning of the procedural program and is computer independent.

The specification is intended as the medium in which analysis of correctness and maintenance is to be performed. The specification can then be automatically forward translated, via "Forward Software Engineering", into a new highly optimized procedural program to conform to any desired computer system. The notion of reverse software engineering is shown in Figure 1.

In a procedural language program, a variable may assume many different values during the course of an execution. In an equational language program, such as MODEL [cccc], a variable may assume one and only value. Therefore, the goal is to a uniquely named variable for each instance that a variable is assigned a value. This is accomplished by mapping each instance of an assignment in the program into an array such that every array element is assigned only one value. The dimensionality of the array is equivalent to the depth the variable is nested plus one. The equations constituting the specification are simply the assignment statements. The correctness of the translation follows from the fact that for each assignment to a variable during the execution of the source program there is an equation that defines a variable in the specification. Thus the source program and the resulting specification are equivalent in that the respective mapped variables have the same value.

The translation consists of eight transformations which are applied in succession. Each of the first five transformations generates a program that is equivalent to the input program. This first group is shown in Figure 2. It starts with the source procedural program and ends with a program that can be directly transformed into equations. In the second group of transformations shown in Figure 3, the final program is translated into equations

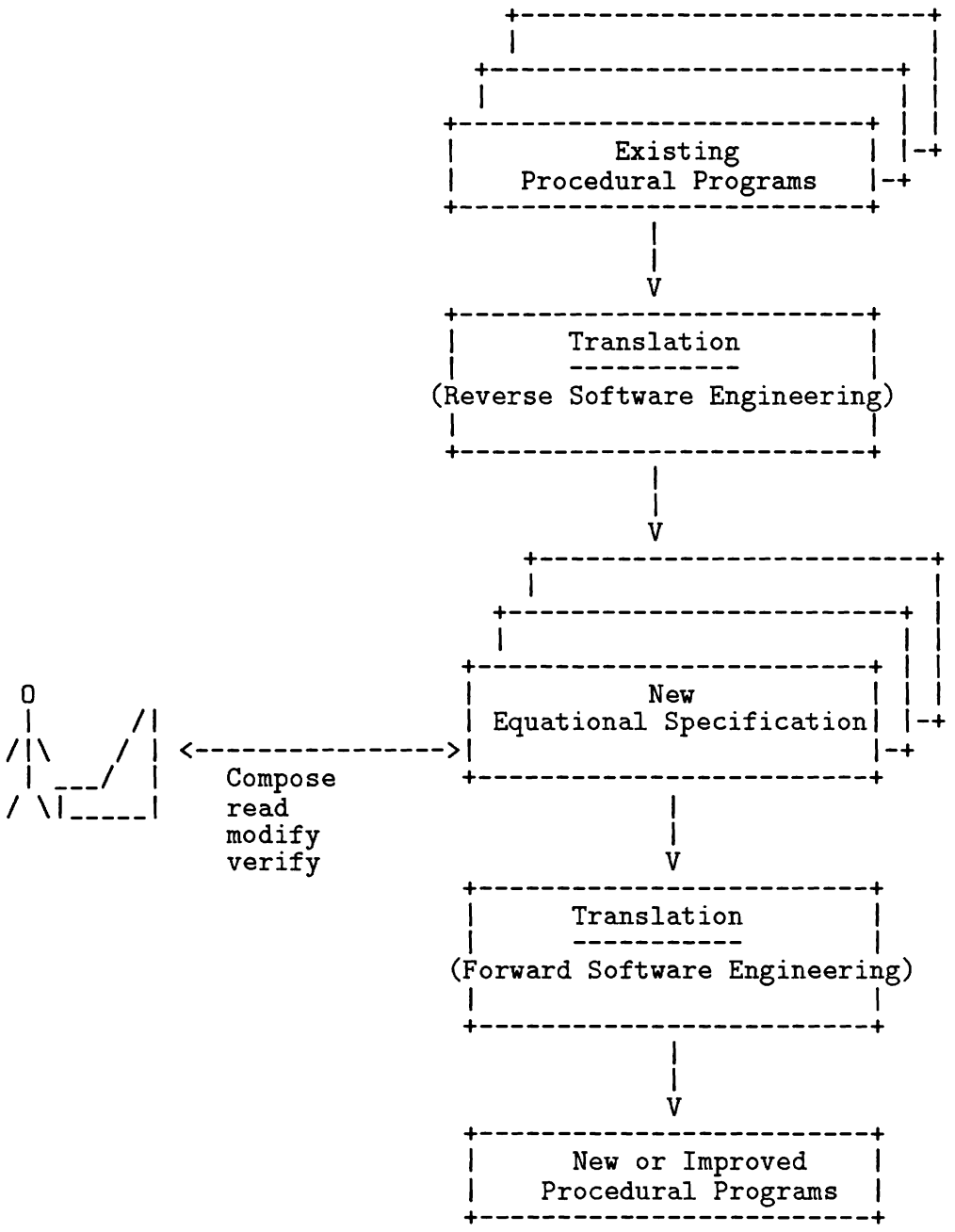


Figure 1: Use of Translations Between Procedural and Equational Languages

and declarations and then simplified. The transformations in the second group operate on equations. They use algebraic identities to collect common factors and use substitutions to simplify the specification.

The following is a brief explanation of the need and purpose of each transformation.

Transformation 1 converts the source program using subroutine calls, *goto* statements, dynamic memory allocations and shared variables into a program using only the basic types of statements.

Transformation 2 builds a tree from the source program such that intermediate nodes represent block statements and leaf nodes represents basic statements. The tree is modified in the succeeding transformations.

Transformation 3 "instruments" the assignments in the program. It nests each *WHILE* and *DO* loop within an *IF* statement block and introduces counters for each *WHILE* and *DO* loops, and for *IF* statements. These counters are used in transformation 4 to identify each instance of an assignment that can be repeated zero or more times. In addition, it eliminates nested *IF* statements by combining conditions used in the nested *IF* statements to simplify the next transformation.

Transformation 4 generates a "single-assignment" program in which a variable is assigned a value by exactly one statement. It first gives distinct names to left hand side (LHS) variables of assignments. It then substitutes variables referenced in expressions with corresponding renamed LHS variables. To simplify the next transformation, if a variable is assigned a value in more than one statement (before LHS renaming) within a loop, it adds extra variable that remembers the last assignment to the variable with the loop.

Transformation 5 generates a "single value variable" program. That is, this transformation completes the conversion of the source program into an equivalent program in which each variable has exactly one value. This is achieved by converting each variable that is assigned multiple times within a loop into an array. The resulting assignments can then be

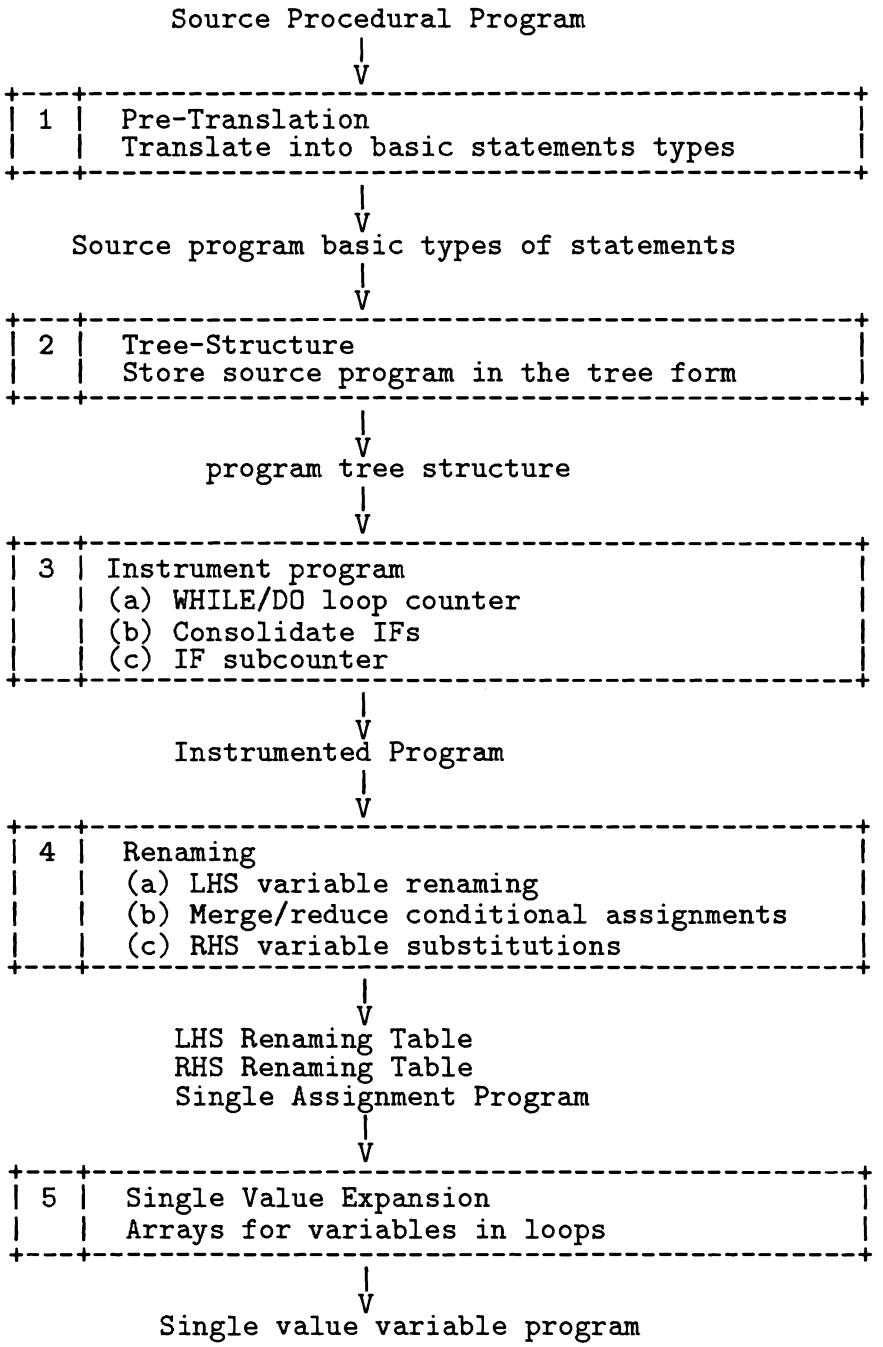


Figure 2: Program Transformations.

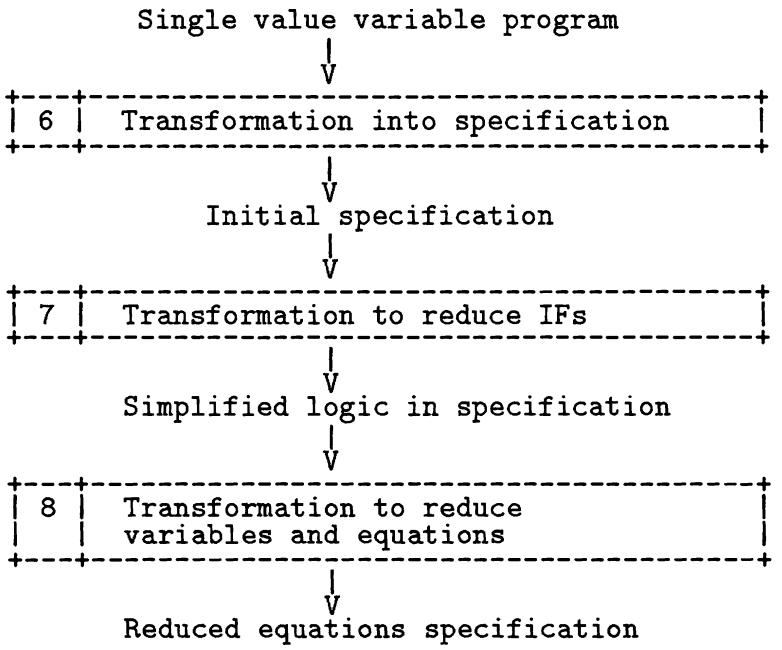


Figure 3: Specification Transformations.

viewed as equations.

Transformation 6 adds a header and declarations to the set of assignments produced previously to make it a MODEL specification. The specification is now longer 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. Transformation 7 identifies common conditions and factors and eliminates multiple definitions of them. In addition, conditions and factors are simplified using identities. Transformation 8 eliminates some variables by substituting their references with defining expressions.

3 MODEL

MODEL is the language used for the object equational specification. This section, taken almost verbatim from [11], describes briefly the syntax and semantics of the basic parts of MODEL.

The benefit of using MODEL is that the present system accepts equational specifications and generates procedural programs in Ada, C and PL/I [2]. In addition, the "specification extension" phase of the MODEL system tolerates omissions in the user provided specification and fills-in missing parts automatically. We take advantage of this feature to simplify the translation.

A MODEL specification consists of a header, declarations and equations. The computation of a specification produces values for variables that make all equations true. The order of equation statements is irrelevant, and thus, may be in any order.

Header The format of a header is shown in Table 1. A header defines the specification type (MODULE, FUNCTION and PROCEDURE), name, inputs (called SOURCE) and outputs (called TARGET). A module denotes a main program. A function has only input parameters and returns a result and a procedure has input, output and update (i.e., both input and output) parameters.

EXTERNAL variables are external variables that are declared, reference or defined in external program entities in an overall software system. External variables may represent structured or elementary variables. They may be used in any kind of program entity (MODULE, FUNCTION and PROCEDURE) and must be listed in the SOURCE and/or TARGET statements.

Declarations The format of a declaration statement is shown in Table 2. Input and output variables including parameters, external variables and variables in I/O devices must be

HEADER:

Name Statements

MODULE: <main procedure name> EXTERNAL(<variable>,...);
FUNCTION: <function name> (<input_parameter>,...) EXTERNAL(<variable>,...)
RESULT(<variable>,...);
PROCEDURE: <subroutine name> (<parameter>,...) EXTERNAL(<variable>,...);

Input/Output Argument Declarations

SOURCE: <input argument or i/o file>,...;
TARGET: <output argument or i/o file>,...;

Table 1: Header Statements in MODEL.

declared. Other (interim) variables can also be declared, but their declarations are optional. If omitted, they are generated automatically in the extension phase of the MODEL system.

Unlike declarations in a procedural language which only identify the structure transferred in each input and output operation, the entire input and output structures must be defined. Each declaration is thus viewed as defining a multi-level tree in a depth-first manner. Each node has a level number, name, repetition (or dimension) size, and data type. Each leaf node must have a primitive data type.

The "rename clause" plays a special role in translating programs using shared variables. Therefore, it is important that we discuss this feature in detail. In numerous cases, such as in the use of shared variables, it is desired that only one memory location be updated and referenced by a number of concurrent programs. This causes a conflict with MODEL as MODEL variables follow the convention in mathematics of being assigned only one value. However, this conflict is resolved through the use of the renaming clause. The renaming clause permits the user of MODEL to specify a mapping of MODEL variables to a single

DECLARATIONS:

Input/Output: Declaration of input/output arguments are mandatory. If it is a structure, the entire input/output data must be declared down to individual data elements and their data types.

1 <structure name><repetitions><type of device>
2 <substructure name><repetitions><structure data type>
⋱
n <elementary variable name> <repetitions>
<primitive data type> [<rename clause>];
<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 file (default),
messages from other processes (tasks),
addressed messages,
random access,
shared memory,
dynamically allocated memory.

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

Table 2: Declarations of Variables in MODEL.

variable of the interfacing program. The rename clause has the following syntax:

<rename clause> ::= RENAME <name>.

In the generated program, the MODEL variable name is renamed to <name>.

Equations Table 3 summarizes the syntax and meaning of *equations*. The left hand side of an equation statement contains only a variable reference. This is the independent variable of the equation. The right hand side is an expression which consists of operations and variables. They have the syntax and meaning of arithmetic or boolean algebra (depending

on the operators).

Variables referenced in equations may be scalars or arrays. An array variable name must be followed by subscript expressions for each of its dimensions, in parenthesis. A variable in some programming languages, as well as in MODEL, may denote an entire tree structure of more elemental variables. Transformation of such structured variables into the basic variables is performed automatically by the MODEL system. If an equation contains a subscripted variable, then an equation is assumed to be repeated for all integer values in the range of 1 to the size of each subscript of the variable. Thus, an equation as well as a variable may be multidimensional. If a subscript expression has the value zero, then the RHS variable in an equation must be preceded by an IF condition preventing use of the variable in such a case.

A specification must include an explicit or implicit definition of each different *size* of a dimension in the variables. The definition is given through a declaration or through an equation. The size of a dimension variable can be denoted by a *control variable.* There are two ways to denote a control variable: by use of the prefix SIZE to denote the number of elements in a dimension of the suffix variable, or by use of the prefix END to denote a boolean vector where each element denotes whether the respective element of the suffix variable is the last one in the dimension. The size of a dimension may be specified to have a zero or positive integer value. If the size is zero, the respective variable is said to be *null.* The equation in which it is the LHS variable is considered to be non-existent.

Several types of operations can be used in expressions. These include arithmetic, logical and string operations. Arithmetic and boolean expressions can be mixed using the *IF-THEN-ELSE* operation, which has the three operands as follows:

IF <boolean expr> THEN <arithmetic expr> ELSE <arithmetic expr>

where ELSE <arithmetic expr> is optional.

MODEL specifications are time invariant. That is, values of variables in a MODEL

EQUATIONS:
<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 sizes 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 > (<subscript expression>,...).

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

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 null) if the size of a LHS variable dimension is zero. The syntax of subscripts is: sub1, sub2,... . These subscripts are local to the equation where they are used. Equations must include qualifying conditions to avoid referencing variables with subscript expression of zero value.

Operations used in expressions include the following:

arithmetic, logical and string operations
if-then-else
built-in or user-defined functions and subroutines

Note: The translator from MODEL to procedural languages tolerates in many cases omissions of definitions of control variables.
There are no order of statements, loop or input/output commands.

Table 3: Equations, Variables, Subscripts and Operations in MODEL.

specification, as in mathematics, are defined by equations only and have no concept of time. All initial values are either declared in the specification or passed in as parameters. In addition to time invariance, a specification has no concept of memory space. Thus it can define an infinite number of variables. (It should be noted that the MODEL compiler does optimize memory usage when generating a program from a specification.)

Further information on MODEL is given in [2].

4 COMMUNICATION BETWEEN CONCURRENT PROCESSES

Concurrent processes can be classified into two categories, independent processes and cooperating processes. A process is *independent* if it cannot affect nor be affected by other processes during its execution [12]. Thus any process that does not share data with any other process is independent. Clearly, the section of code that execute in an independent process can be thought of as a sequential subroutine. Thus translation of such code into equations is equivalent to the translation of any other sequential subroutine.

The interesting case involves cooperating processes. A process is *cooperating* if it can affect or be affected by other processes during its execution [12]. Thus any process that share data with and/or send messages to other processes is a cooperating process.

There is only one real difference between concurrent programs and sequential programs. This is that processes in concurrent programs may execute in parallel and communicate with other executing processes. Thus the problem of extending the translation algorithm of [11] to include concurrent programs, is really the problem of translating the features in concurrent programs that facilitate interprocess communication.

4.1 Communication Through Messages

Communication through messages can be either direct or indirect. In *direct communication*, each process that wants to send a message must explicitly name the recipient of the message. Likewise, each process that wants to receive a message must explicitly name the sender of the message. Communication in such a scheme is carried out by means of the following primitives or a variant there of:

send(P,*message*) - send *message* to process P,

receive(P,message) - receive *message* from process P.

In *indirect communication*, processes communicate via an intermediate repository, commonly known as a mailbox. A mailbox is abstractly viewed as an object into which messages may be placed by processes and from which messages may be removed [12]. Here, the primitives are defined as follows:

send(M,*message*) - send a *message* to mailbox M,
receive(M,message) - receive a *message* from mailbox M.

A process may communicate with another process only if they share at least one mailbox. Each mailbox has a unique identification.

The MODEL system supports indirect communication via mailboxes. It views mailboxes as external I/O devices, i.e. external to the specification, to and from which messages may be sent and received. Although, mailboxes are viewed as external devices they do not physically exist.

To facilitate this approach, MODEL makes available to the user the two file types MAIL and POST. A MAIL file is used to communicate with other processes. It may receive from one or more other processes but can only be read by one process. POST files are used when one process wishes to send a message to many other processes. It differs from a MAIL file in that it includes the address of its destination as a key.

To send a message in MODEL one need only define a value for a variable in a MAIL file. Similarly, to receive a message one need only reference a variable in a MAIL file. This is illustrated in Figure 4. The first equation sends a message by assigning a value to *msg_q*. The second equation receives a message by simply referencing *msg_p*. Thus these equations are equivalent to the execution of the primitives send(mailbox_q,"send message") and receive(mailbox_p,msg_p), respectively.

```
PROCEDURE: Example1 EXTERNAL(mailbox_p,mailbox_q);
SOURCE: mailbox_p;
TARGET: mailbox_q;

1 mailbox_p IS FILE ORG=MAIL,
  2 message_p IS RECORD,
    3 msg_p IS FIELD (char(20));

1 mailbox_q IS FILE ORG=MAIL,
  2 message_q IS RECORD,
    3 msg_q IS FIELD (char(20));

msg_q = "send message";
msg = msg_p;
```

Figure 4: Example Specification using Mailboxes

4.2 Communication Through Shared Memory

Shared variables are used by concurrent programs to facilitate communication, in one way or another, between concurrent processes. Communication is accomplished by assigning values to and referencing from shared variables. In the course of a computation, a shared variable may be assigned many values by different processes.

MODEL follows the mathematical convention that a variable can assume only one value. Thus all values generated must be assigned to a separate variable in a MODEL specification. However, in the case of shared variables, it is desirable to update only one memory location. The renaming clause in MODEL, discussed in Section 3, permits us to show such a correspondence.

The use of the renaming clause is illustrated in Figure 5. The procedure Example2 tests the value of $flag1$, and if it is FALSE, sets $flag2$ to TRUE. From the view of the specification, $flag1$ and $flag2$ are two separate local variables. But to the external world, procedure Example2 operates on the shared variable $flag$.

```
PROCEDURE: Example2 EXTERNAL(sharedin, sharedout);
SOURCE: sharedin;
TARGET: sharedout;

1 sharedin IS FILE ORG=SHARED,
  2 rec1(*) IS RECORD,
    3 flag1 IS FIELD (bit(1)) RENAME flag;

1 sharedout IS FILE ORG=SHARED,
  2 rec2 IS RECORD,
    3 flag2 IS FIELD (bit(1)) RENAME flag;

END.flag1(sub1) = flag1(sub1)=FALSE;

flag2 = DEPENDS_ON(flag1(sub1));
flag2 = TRUE;
```

Figure 5: Example Specification using Shared Variables

The DEPENDS_ON is a MODEL built in function that defines the evaluation precedence between two variables. Thus the second equation merely states that flag1 is to be evaluated before flag2.

Shared memory is viewed as an external I/O device by MODEL. Thus all shared variables are declared in SHARED files. SHARED files are external random access devices where its contents are seen by equations of a specification through templates. These templates allow an equation to only see the shared variables it updates or references. A template shows either input variables or an output variable but not both. Figure 6 shows this correspondence.

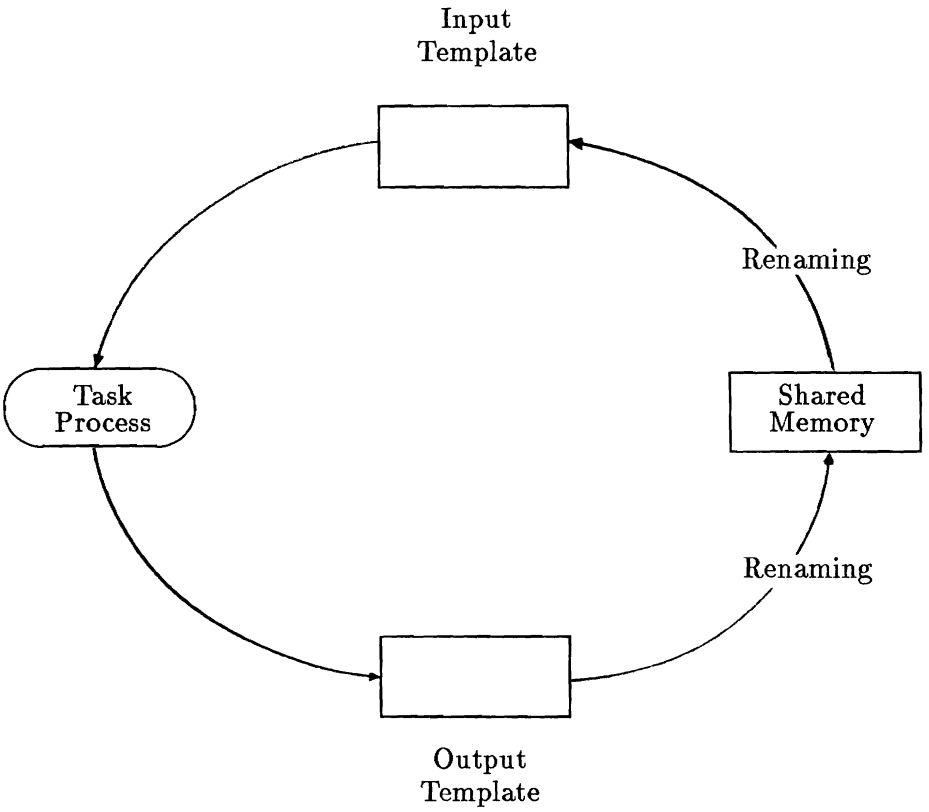


Figure 6: MODEL's View of Shared Memory

5 EXAMPLE OF TRANSFORMATION

The basic differences between concurrent programs and sequential programs are the features introduced in concurrent programs to handle interprocess communication during parallel execution of processes. For without interprocess communication, there is no real distinction between concurrent and sequential programs. Furthermore, each concurrent process is nothing more then a sequential program. So the problem of extending the RSE algorithm to translate concurrent programs is, as stated previously, really the problem of extending the translation algorithm to handle statements that facilitate interprocess communication between concurrent processes. Therefore, it is not surprising that the translation algorithm for concurrent programs is just the translation algorithm for sequential programs extended to include the translation of those features that facilitate interprocess communication.

As indicated by Section 4, there are numerous schemes for allowing interprocess communication. However, to present a translation algorithm for all possible schemes would go far beyond the bounds of this paper. We therefore limit ourselves to the translation of programs that use only shared variables as a means of communication between concurrent processes. [Note however that MODEL has features to specify and generate programs that exchange messages between processes. An example was given in Section 4.1.]

The translation algorithm is described with the aid of an example. For this we have selected the classic algorithm presented by Dekker as a software solution to the critical section problem [3]. The routines, Enter() and Leave(), that implement Dekker's algorithm are given in Figure 7. We have selected this algorithm because it shows many of the features of the reverse translation algorithm. It also shows the less attractive side of our solution. It was deliberately chosen to curb any claim of our fixing the example to our benefit.

Figure 8 shows how two processes use the procedures Enter() and Leave() to achieve mutual exclusion in their critical section. When process *pi* wishes to enter its critical section, it blocks process *pj* from entering *pj*'s critical section by executing Enter(i). When *pi* is

finished, it then unblocks the other process by executing Leave(i).

Given two processes $p1$ and $p2$, each with a critical section, Dekker's algorithm insures mutual exclusion, progress and bounded wait through the use of the shared variables c[1], c[2] and turn. c[1] and c[2] are booleans and turn is an integer that assumes either 1 or 2.

Observe that c[i] may only be changed by pi. Whenever process pi wants to enter its critical section, it sets c[i] to TRUE. A process pi may enter its critical section only if c[i]=TRUE and c[j]=FALSE. In the event that both processes wish to enter at the same time, the variable turn dictates which process enters first and which process waits. pi, upon leaving the critical section, sets turn to the index of the other process.

Despite its brevity, Dekker's algorithm is very complex as it has numerous side effects that are not apparent superfically. The transformations given below will make these side effects explicit in the object specification.

A non-rigorous proof of correctness of Dekker's algorithm, based on the procedural language program, is given in [3]. Using the equational language specifications, a more rigorous proof may be easily given. In Section 6, we present an approach based on the equational specification to proving correctness. It also "explains" the function of the procedure Enter() by enumerating the cases that must be considered in proving mutual exclusion, progress and bounded wait.

The following subsections describe the eight transformations that make up our algorithm. Because the main focus of this paper is to show how concurrent programs, i.e. interprocess communications, are translated and the algorithm for sequential programs has already been given in [11], we discuss in detail only those parts of the algorithm that pertain to concurrency. Other parts of the algorithm are summarized. Readers wishing more detail may consult [11].

For our example, the reader should not be concerned with the choice of the procedural programming language used, as any procedural language program can be pre-translated to

```
01 PROCEDURE Enter (i: integer);
02   j: integer;
03
04 begin
05   j = 3-i;
06   c[i] = true;
07   while (c[j]) do
08     if turn = j then
09       c[i] = false;
10       while (turn = j) do
11       endwhile
12       c[i] = true;
13     endif
14   endwhile
15 end
16
17
18 PROCEDURE Leave (i: integer);
19
20
21 begin
22   turn = 3-i;
23   c[i] = false;
24 end
```

Figure 7: Dekker's Algorithm In Procedural Form

```
c[1]=c[2]=FALSE: SHARD BOOLEAN;
turn=2: SHARED INTEGER;

COBEGIN
   PROCESS p1:
      LOOP
           .
           .
           .
         CALL Enter(1);
         <critical section>
         CALL Leave(1);
           .
           .
           .
      ENDLOOP
||
   PROCESS p2:
      LOOP
           .
           .
           .
         CALL Enter(2);
         <critical section>
         CALL Leave(2);
           .
           .
           .
      ENDLOOP
COEND
```

Figure 8: The Critical Section Problem

have only the basic types of statements [11].

We note that since Enter() and Leave() are procedures, they are translated independently into separate equational specifications. We also note that because the translation algorithm makes use of line numbers in generating new names for variables, will include line numbers in our figures for completeness.

5.1 Pre-Translation

Transformation one converts the source program into a program that is composed of only the basic types of statements common to most procedural languages. These basic statements are input/output (read, write), assignment, *DO*, *WHILE* and *IF* statements and declaration of variables. Discussions of translating numerous "non-basic" statements to their corresponding basic statements are given in [1] and [11]. We discuss here only the transformation of statements that refer to shared variables.

As mentioned in Section 4.2, shared memory is envisioned as an external I/O device to which read and write operations are performed. Thus for each occurrence of a shared variable referenced on the RHS of a statement, we insert immediately before the statement a pseudo-read operation on the external device. We likewise insert a pseudo-write operation on the external device immediately after each occurrence of a shared variable on the LHS of an assignment statement. All shared variables are then newly declared as local variables. These local variables will later be mapped to their respective shared variables. This translation is illustrated in Figure 9.

Although a shared variable may be assigned many different values by many processes, when it is referenced the variable has only the most recent value. Therefore, the transformation of shared variables in this way does not alter the meaning of the program.

The benefit of this transformation is that the program is now void of shared variables and their side effects, i.e. all variables are local and assigned values only by statements

⟨shared var⟩ = ⟨expression⟩	⟨var⟩ = ⟨expression⟩ WRITE ⟨var⟩ TO ⟨shared file⟩

(a) Translation of an Assignment to a Shared Variable

⟨var1⟩ = f(⟨shared var⟩)	READ ⟨var2⟩ FROM ⟨shared file⟩ ⟨var1⟩ := f(⟨var2⟩)
⟨condition(shared var)⟩)	READ ⟨var⟩ FROM ⟨shared file⟩ ⟨condition(var)⟩)

(b) Translation of a Reference to a Shared Variable

Figure 9: Translation of Shared Variables

within the process.

The result of Transformation 1 is show in Figure 10. The EXTERNAL statement introduced in the new program merely states that the filenames in its parameter are external devices. This statement is equivalent to the MODEL EXTERNAL statement discussed in Section 3.

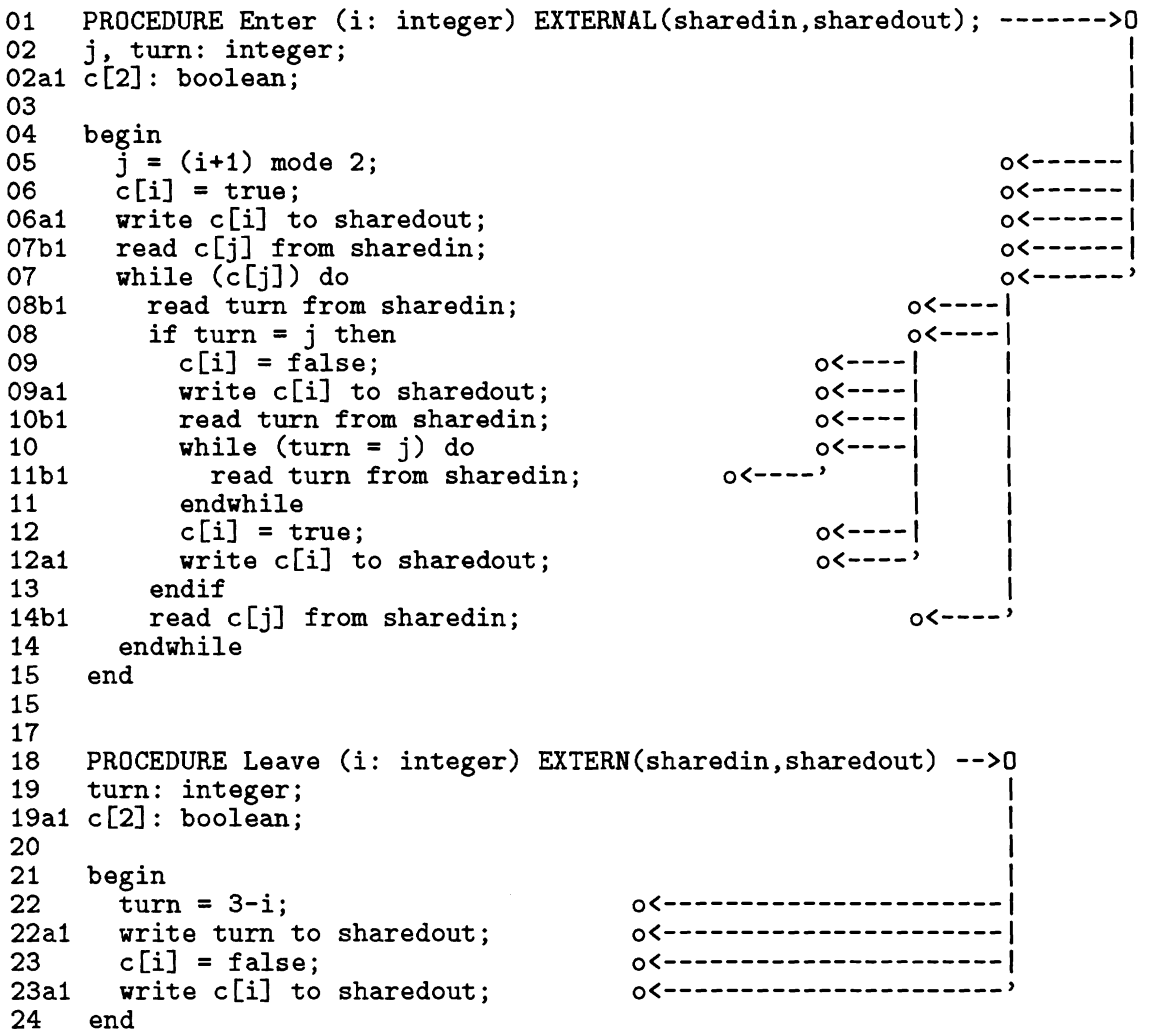


Figure 10: Pre-translated Program with Program Tree

5.2 Program Tree

The right side of Figure 10 shows the program tree for our sample program. The tree is constructed from the source program such that intermediate nodes represent block statements and leaf nodes represent basic statements. The remaining transformations operate on this tree. In fact, the first six transformations can be expressed as operations on the tree. However, for ease of reading we display the results of each transformation in non-tree format.

5.3 Instrumentation

We perform three depth-first traversals on the program tree. On the first traversal, we instrument each WHILE loop node as illustrated in Figure 11 and each DO loop node as illustrated in Figure 12. That is, we first nest each loop block within an IF block, the effect of which is to push the loop node further from the root. This instrumentation is necessary to account for and handle the case that the body of the loop is not executed at all as a result of failing the loop condition on the first try. We then introduce a loop counter, *subi*, and a variable, *sizei* for a DO loop and *endi* for a WHILE loop, that defines the number of iterations. The reader should not that if the IF condition is satisfied, the nested loop will always be executed at least once. The objective of these transformations is to provide the variables that index each variable defined in a loop and obtain a count of such definitions.

On the second traversal, we instrument each IF node as shown in Figures 13 and 14. If the distance to the root is one, then we transform the IF node into a DO loop. Note that the DO loop is executed either zero or once. The reason for this instrumentation is largely for uniformity in the tree, consisting of loop blocks with IF blocks nested in them.

If the distance to the root node is greater then one, then we introduce in each loop sublinear subscripts for each conditional block.

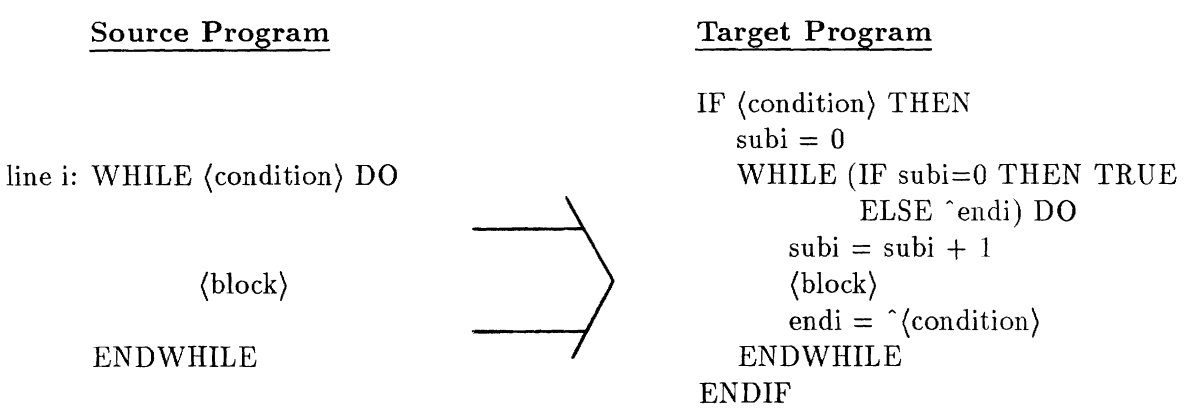


Figure 11: Transforming WHILE Loop

A sublinear subscript denotes the index of the variables defined in the IF block, i.e. conditional block, and is viewed as a vector with an element for each subscript of the immediately nesting loop. A sublinear subscript is called *slk*, where k is the statement number of the respective THEN or ELSE block of the IF statement. The condition itself is called *slcj*, where j is the statement number of the IF condition. *slcj* and *slk* are functions of their respective loop counter *subi*. Note that *subi* may itself be a sublinear subscript. The values of the sublinear subscripts are defined by the function sublinear(). Given a sublinear condition slcj, sublinear subscript slk, and subscript subi, the following relationship holds for the sublinear function:

sublinear(slcj,slk,subi) = IF subi=1 THEN IF slcj THEN 1 ELSE 0
ELSE IF slcj THEN slk+1 ELSE slk.

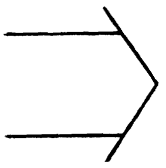
Thus a sublinear subscript is incremented only if the sublinear condition is true.

Source Program

line i: DO I=⟨exp1⟩TO⟨exp2⟩,⟨exp3⟩

⟨block⟩

ENDDO



Target Program

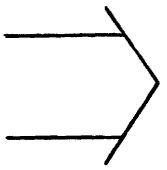
IF ⟨exp2⟩ ≥ ⟨exp1⟩ THEN
sizei = (⟨exp2⟩-⟨exp1⟩+⟨exp3⟩)/⟨exp3⟩
DO subi=1 TO sizei
I = IF subi=1 THEN exp1
ELSE I+exp3
⟨block⟩
ENDDO
ENDIF

Note: ⟨exp1⟩, ⟨exp2⟩ and ⟨exp3⟩ are integer expressions where ⟨exp3⟩>0 and they do not depend on the block.

Figure 12: Transforming DO Loop

Source Program

line i: IF ⟨condition⟩ THEN
⟨block⟩
line j: ELSE
⟨block⟩
ENDIF



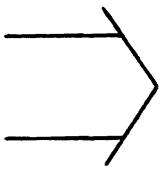
Target Program

sizei = IF ⟨condition⟩ THEN 1 ELSE 0
sizej = IF ^⟨condition⟩ THEN 1 ELSE 0
DO subi=1 TO sizei
⟨block⟩
ENDDO
DO subj=1 TO sizej
⟨block⟩
ENDDO

Figure 13: Transforming Non-Loop-Nested IF

Source Program

```
line j: IF ⟨condition⟩ THEN
            ⟨block⟩
line k: ELSE
            ⟨block⟩
        ENDIF
```



Target Program

```
slcj = ⟨condition⟩
prev.slj = IF subi>1 THEN slj
prev.slk = IF subi>1 THEN slk
slj = sublinear(slcj,slj,subi)
slk = sublinear(^slcj,slk,subi)
IF slcj THEN
    ⟨block⟩
ELSE
    ⟨block⟩
ENDIF
```

Notes: subi is the subscript of the loop that directly nests the IF.

prev.slj and prev.slk are saved values of sublinear subscripts slj, slk previous to their updating (in the subi-1 iteration).

Figure 14: Transforming Nested IF

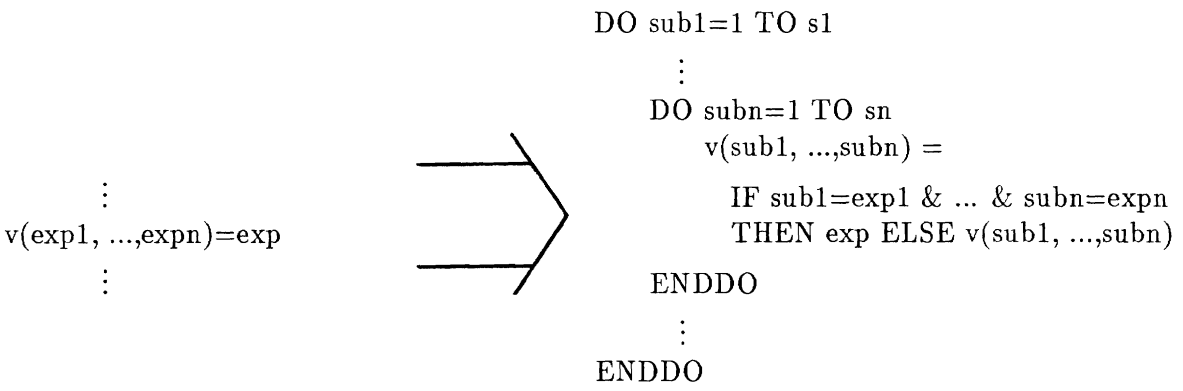


Figure 15: Transforming Assignments to a Program Array Variable

On the third traversal, each statement that assigns a value to a program array variable is instrumented as illustrated in Figure 15. That is, each statement is nested in a series of loops, where the number of loops is equivalent to the dimension of the array variable. The effect is to update all other elements of the array with their respective old values whenever a given element of the array is assigned a value. How this benefits our translation may seem unclear at this point but suffice it to say that this instrumentation greatly simplifies subscript determination in transformation 5. For a more detailed explanation, the reader may consult [11].

This instrumentation of arrays, however, is unnecessary for arrays mapped to shared arrays since each element of the array is updated (through a read or an assignment operation) before its use. Thus in the instrumented program, array variable c is not nested in a loop. All references to an element of c is preceded by an update to that element.

Figure 16 shows the instrumented program.

```
01    PROCEDURE Enter (i: integer) EXTERNAL(sharedin,sharedout);
02    j, turn: integer;
02a1  c[2]: boolean;
03
04    begin
05      j = (i+1) mode 2;
06      c[i] = true;
06a1    write c[i] to sharedout;
07b1    read c[j] from sharedin;
07b2    size7 = if c[j] then 1 else 0;
07b3    do sub1=1 to size7
07b4      sub2 = 0;
07        while (if sub2=0 then TRUE else ^end7) do
07a1        sub2 = sub2 + 1;
08b1        read turn from sharedin;
08b2        slc8 = turn=j;
08b3        prev.sl8 = if sub2>1 then sl8;
08b4        sl8 = sublinear(slc8,sl8,sub2);
08          if slc8 then
09            c[i] = false;
09a1          write c[i] to sharedout;
10b1          read turn from sharedin;
10b2          slc10 = turn=j;
10b3          prev.sl10 = if sl8>1 then sl10;
10b4          sl10 = sublinear(slc10,sl10,sl8);
10b5          if slc10 then
10b6            sub3 = 0;
10              while (if sub3=0 then TRUE else ^end10) do
11b1              read turn from sharedin;
11b2              end10 = ^(turn=j);
11              endwhile
11a1          endif
12            c[i] = true;
12a1         write c[i] to sharedout;
13          endif
14b1        read c[j] from sharedin;
14b2        end7 = ^(c[j]);
14        endwhile
14a1    enddo
15    end
15
17
18    PROCEDURE Leave (i: integer) EXTERN(sharedin,sharedout);
19    turn: integer;
19a1  c[2]: boolean;
20
21    begin
22      turn = 3-i;
22a1    write turn to sharedout;
23      c[i] = false;
23a1    write c[i] to sharedout;
24    end
```

Figure 16: Instrumented Program

5.4 Renaming for Single Assignment

This transformation transforms the output program of Transformation 3 into a single assignment program. A *single assignment program* is a program where each of its variables is assigned a value by one assignment statement only. The transformation is accomplished by renaming each LHS variable in an assignment statement to have a unique name .

This transformation has three phases: (1) LHS renaming, (2) addition of localizing loop variables and (3) RHS renaming.

LHS renaming This phase renames the variables on the LHS of each assignment statement (for the purpose of our algorithm, a READ statement is considered to be an assignment statement). This is accomplished by simply appending the line number of the statement to the LHS variable name. This results in a distinct LHS variable for each assignment statement. In this way the single assignment rule is applied to the program.

Figure 17 shows a table that summarizes the LHS renaming. It shows for each LHS variable the statement address, i.e. line number, it appears in and its new name for that occurrence. The subscripts column is actually generated in the next transformation and will be used to achieve our goal of having each variable assume only one value.

Localizing Loop Variables This phase is concerned with adding assignments to localize loop variables. It merges conditional assignments to the same variable to reduce the number of LHS variables that must be considered for RHS referencing. It also defines for each LHS variable in a loop two types of *dimension-reduction* variables: a boolean variable that denotes whether the variable is null, i.e. is not assigned a value, and a variable that represents the last value of the variable at the termination of each loop provide it is not null. These dimension-reduction variables facilitate the generation of substituting expressions for RHS variables.

As will be noted, localizing loop variables will not be necessary for variables declared in the pre-translation of shared variables as on every reference we read a separate variable.

RHS variables This phase generates the expressions for substituting renamed LHS variables for variables on the RHS of each assignment statement. The tree is scanned to find for each RHS variable the same named LHS variables that have been assigned values preceding the RHS reference. In the case of variables introduced by the pre-translation of shared variables this is always the assignment statement (or read statement) that immediately precedes its reference. For this reason dimension-reduction variables are not needed. As an example refer to Figure 19, line 14b1 is a read statement with c_14[j_5] as the LHS variable. So in line 14b2, the RHS variable c[j_5] is renamed to c_14[j_5].

This renaming is also illustrated in the RHS renaming table of Figure 18. It shows for each RHS occurrence of a variable its statement address and the substituting expression. For original local variables, i.e. those not as a result of the pre-translation of shared variables, the substituting expression may be simple or complex. But for those variables declared as a result of the pre-translation, the substituting expression is always simple as evidenced in the table.

Figure 19 shows the resulting single assignment program.

var	statement address	renamed as	subscripts
j	5	j_5	scalar
c	6	c_5	i
	7b1	c_7	j_5
	9	c_9	sub1,sl8[sub1,sub2],i
	12	c_12	sub1,sl8[sub1,sub2],i
	14b1	c_14	sub1,sl8[sub1,sub2],j_5
	23	c_23	i
turn	8b1	turn_8	sub1,sub2
	10b1	turn_10	sub1,sl8[sub1,sub2]
	11b1	turn_11	sub1,sl10[sub1,sl8[sub1,sub2]],sub3
	22	turn_22	scalar
size7	7b2	size7	scalar
slc8	8b2	slc8	sub1,sub2
prev.sl8	8b3	prev.sl8	sub1,sub2
sl8	8b4	sl8	sub1,sub2
slc10	10b2	slc10	sub1,sl8[sub1,sub2]
prev.sl10	10b3	prev.sl10	sub1,sl8[sub1,sub2]
sl10	10b4	sl10	sub1,sl8[sub1,sub2]
end10	11b2	end10	sub1,sl10[sub1,sl8[sub1,sub2]],sub3
end7	14b2	end7	sub1,sub2

Figure 17: LHS Renaming Table

var	statement address	substituting expression
j	7b1	j_5
	7b2	j_5
	8b2	j_5
	10b2	j_5
	11b2	j_5
	14b1	j_5
	14b2	j_5
c	6a1	c_6
	7b2	c_7
	9a1	c_9
	12a1	c_12
	14b1	c_14
	23a1	c_23
turn	8b2	turn_8
	10b2	turn_10
	11b2	turn_11
	22a1	turn_22
size7	7b3	size7
slc8	8b4	slc8
	8	slc8
sl8	8b3	sl8
	8b4	sl8
	10b3	sl8
	10b4	sl8
slc10	10b4	slc10
	10b5	slc10
sl10	10b3	sl10
	10b4	sl10
end7	7	end7
end10	10	end10

Figure 18: RHS Renaming Table

```
01    PROCEDURE Enter (i: integer) EXTERNAL(sharedin,sharedout);
02    j_5, turn_8, turn_10, turn_11: integer;
02a1  c_5[2], c_7[2], c_9[2],c_12[2], c_14[2]: integer;
03
04    begin
05      j_5 = (i+1) mode 2;
06      c_6[i] = true;
06a1    write c_6[i] to sharedout;
07b1    read c_7[j_5] from sharedin;
07b2    size7 = if c_7[j_5] then 1 else 0;
07b3    do sub1=1 to size7
07b4      sub2 = 0;
07        while (if sub2=0 then TRUE else ^end7) do
07a1        sub2 = sub2 + 1;
08b1        read turn_8 from sharedin;
08b2        slc8 = turn_8=j_5;
08b3        prev.sl8 = if sub2>1 then sl8;
08b4        sl8 = sublinear(slc8,slc8,sub2);
08          if slc8 then
09            c_9[i] = false;
09a1          write c_9[i] to sharedout;
10b1          read turn_10 from sharedin;
10b2          slc10 = turn_10=j_5;
10b3          prev.sl10 = if sl8>1 then sl10;
10b4          sl10 = sublinear(slc10,sl10,sl8);
10b5          if slc10 then
10b6            sub3 = 0;
10              while (if sub3=0 then TRUE else ^end10) do
11b1              read turn_11 from sharedin;
11b2              end10 = ^(turn_11=j_5);
11              endwhile
11a1          endif
12            c_12[i] = true;
12a1          write c_12[i] to sharedout;
13          endif
14b1        read c_14[j_5] from sharedin;
14b2        end7 = ^(c_14[j_5]);
14        endwhile
14a1    enddo
15    end
15
17
18    PROCEDURE Leave (i: integer) EXTERN(sharedin,sharedout);
19    turn_22: integer;
19a1  c_23[2]: boolean;
20
21    begin
22      turn_22 = 3-i;
22a1    write turn_22 to sharedout;
23      c_23[i] = false;
23a1    write c_23[i] to sharedout;
24    end
```

Figure 19: Single Assignment Program

5.5 Single Value Assignment

To achieve single value assignment, the declaration of each LHS variable in a loop is modified to increase its dimension by one for each level the variable is nested. If the size of a dimension is not know at this point, an asterisk (*) is used to denote an arbitrary size. The sizes for these dimensions are computed at run time by the assignments to the *size* and *end* variables. The respective loop counters are then inserted as subscripts to these variables. The order of the subscripts reflect the order of the nesting loops. The subscripts, therefore, represents the loops the variable is nested in. If a variable is nested in an IF block, the sublinear subscript defined for that IF block is used in place of the loop counter. Note that this rule for substituting a sublinear subscript for a loop counter is a recursive rule, as a sublinear subscript is itself a loop counter.

The subscripts of each variable is shown in the subscripts column of Figure 17 and the program with the added subscripts is shown in Figure 20.

This completes our transformation from procedural language program to procedural language program. The next transformation extracts the equations from the single value variable program. The remaining transformations are concerned with simplifying the initial equations.

5.6 Initial Specification

Figures 22, 23 and 24 are the results of transformation 6. The equations are merely the assignment statements of the program in Figure 20 with MODEL key words, in capital letters, substituted. They include the header and variable declarations necessary in MODEL.

The header and I/O declarations in a MODEL specification establish the interface with the external environment. The procedure name identifies itself to the external en-

```
01   PROCEDURE Enter (i: integer) EXTERNAL(sharedin,sharedout);
02   j_5, turn_8[*,*], turn_10[*,*], turn_11[*,*,*]: integer;
02a1 c_5[*,*,2], c_7[*,*,2], c_9[*,*,2], c_12[*,*,2], c_14[*,*,2]: integer;
03
04   begin
05     j_5 = (i+1) mode 2;
06     c_6[i] = true;
06a1   write c_6[i] to sharedout;
07b1   read c_7[j_5] from sharedin;
07b2   size7 = if c_7[j_5] then 1 else 0;
07b3   do sub1=1 to size7
07b4     sub2 = 0;
07       while (if sub2=0 then TRUE else ^end7) do
07a1       sub2 = sub2 + 1;
08b1       read turn_8[sub1,sub2] from sharedin;
08b2       slc8[sub1,sub2] = turn_8[sub1,sub2]=j_5;
08b4       sl8[sub1,sub2] = sublinear(slc8[sub1,sub2],slc8[sub1,sub2],sub2);
08         if slc8[sub1,sub2] then
09           c_9[sub1,sl8[sub1,sub2],i] = false;
09a1         write c_9[sub1,sl8[sub1,sub2],i] to sharedout;
10b1         read turn_10[sub1,sl8[sub1,sub2]] from sharedin;
10b2         slc10[sub1,sl8[sub1,sub2]] = turn_10[sub1,sl8[sub1,sub2]]=j_5;
10b4         sl10[sub1,sl8[sub1,sub2]] = sublinear(slc10[sub1,sl8[sub1,sub2]],
                                                   sl10[sub1,sl8[sub1,sub2]],
                                                   sl8[sub1,sub2]);
10b5         if slc10[sub1,sl8[sub1,sub2]] then
10b6           sub3 = 0;
10             while (if sub3=0 then TRUE
                       else ^end10[sub1,sl8[sub1,sub2],sub3]) do
11b1             read turn_11[sub1,sl8[sub1,sub2],sub3] from sharedin;
11b2             end10 = ^(turn_11[sub1,sl8[sub1,sub2],sub3]1=j_5);
11             endwhile
11a1         endif
12           c_12[sub1,sl8[sub1,sub2],i] = true;
12a1         write c_12[sub1,sl8[sub1,sub2],i] to sharedout;
13         endif
14b1       read c_14[sub1,sub2,j_5] from sharedin;
14b2       end7 = ^(c_14[sub1,sub2,j_5]);
14       endwhile
14a1   enddo
15   end
15
17
18   PROCEDURE Leave (i) EXTERN(sharedin,sharedout);
19   turn_22: integer;
19a1 c_13[2]: boolean;
20
21   begin
22     turn_22 = 3-i;
22a1   write turn_22 to sharedout;
23     c_23[i] = false;
23a1   write c_23[i] to sharedout;
24   end
```

Figure 20: Single Value Variables Program

vironment and the EXTERNAL parameters allow the sharing of memory with other processes. All structures in the EXTERNAL parameters must be listed in either or both the SOURCE and TARGET statements and defined via the declaration statement. The statement ORG=SHARED states that the file, i.e. memory, is shared with other tasks. The RENAME clause maps the structure being declared to the structure following the RENAME clause.

Variables in SHARED files are structured as they appear in a depth-first search of the program tree. Declarations of SHARED files needed to describe the I/O for renamed variables are determined as follows. First, renamed variables are first grouped into input and output variables. Then, each of the two groups are further separated into smaller groups such that variables in the same group have the same subscripts. Each of these groups corresponds to a SHARED file in our specification.

The control variables SIZE.sl8(*,*), END.slc8(sub1,sub2) and END.turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3) denotes the size of the first (sub1), second (sub2) and third (sub3) dimensions, respectively.

An equation is such that its dependencies on other variables are explicitly given in the equation. Thus the order of evaluation of an equation is determined by its dependencies. However, an equation's dependencies does not always determine timing precedences between shared variables.

Shared variables are often placed in a procedural program under the assumption that they will be evaluated in a certain order. That is to say, a time dependency exists in concurrent programs that use shared variables. To change the order of evaluation of a shared variable will most certainly change the meaning of the program. Thus, it is crucial that this order of evaluation be preserved. Equations, however, are time invariant and do not express timing dependencies. Therefore, another mechanism is introduced to enforce this dependency. This is the DEPENDS_ON() function. An equation of the form

x = DEPENDS_ON(y)

indicates a precedence of y over x. That is, y is to be evaluated before x.

Figure 21 illustrates by example how MODEL views shared memory. Each of the three shared files of Enter() are represented in the figure as a separate tree structure. The shape of each tree (the solid edges) is defined by the declaration of the file it represents, which is identified by the root label. The top two trees represents the two SOURCE files sharedin1 and sharedin2. The bottom tree represents the TARGET file sharedout. The combination of the solid and dotted directed edges between nodes within the same tree indicates the order of reading (for SOURCE) or writing (for TARGET) defined by a depth-first traversal of the tree (note that nodes with an asterisk may be visited repeatedly). The dotted directed edges between trees represent precedences that exist but are not determined by either data dependencies or file declarations. The algorithm to determine these edges is given below.

The following algorithm determines the directed edges between trees. For each statement that references or assigns to variable x renamed to a shared variable, we scan the program tree in reverse depth-first order looking for the first (nearest) preceding statement with an occurrence of any other variable y renamed to a shared variable. (This excludes READ and WRITE statements as they are not part of the specification.) If a precedence edge, dotted or solid, does not already exist between the two variables, we than add a dotted directed edge from y to x. This is accomplished by adding to the specification the statement:

x = DEPENDS_ON(y).

If the nearest statement is an IF statement then we may arbitrarily pick the last such y in the THEN block or the ELSE block. The resulting edges along with their respective statement are shown in Figure 21 as dotted directed edges between trees.

Since the thrust of this research has been on the ease of user understanding of the algorithm that underlies a program, it is important to note that a diagram such as in Figure 21 can be generated automatically and is important to our "explaining" goal.

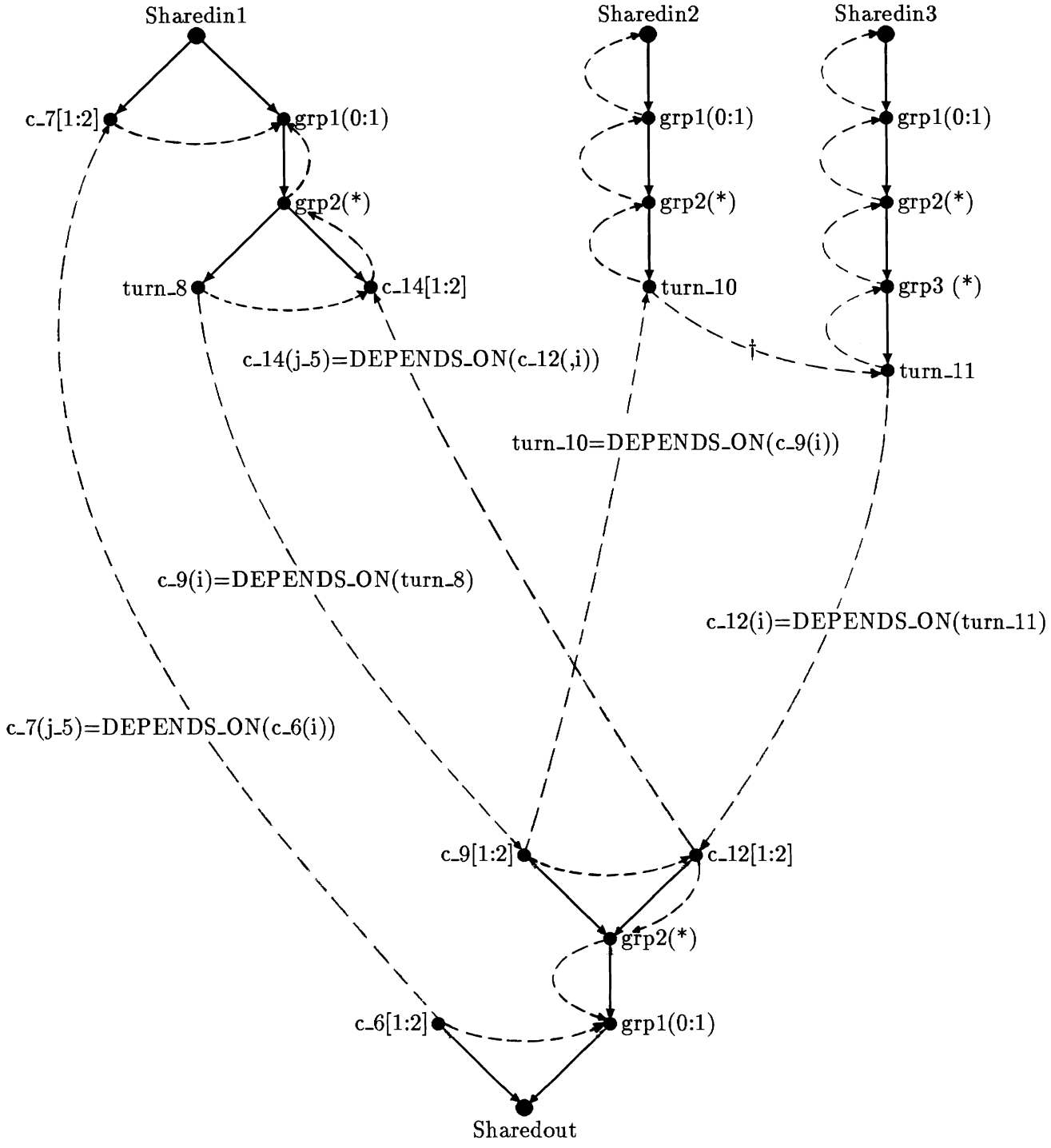


Figure 21: Dataflow Diagram of the Shared Structures Sharedin1, Sharedin2 and Sharedout

5.7 Final Specification

The 7-th and 8-th transformation simplifies the logic and reduces the equations, respectively. Because of the simplicity of the way variables renamed to shared variables are transformed, the equations are already in a reduced form. Thus Transformations 7 and 8 have little or no effect on these equations.

Since no further simplification can be performed on our initial specifications, they are also our final specifications.

5.8 Comments

At first glance the reader may be surprised at the length of the final specification for Enter(). However, the reader is reminded of our goal of making explicit in the final specification the subtle attributes in the source program. This, we feel, is one of the important result.

Procedural language programs often look deceivingly simple. A user of a procedural language program may feel that they completely understand what a program does. When in reality, they may not have considered the many special cases that are often hidden in a program. As any programmer knows, it is these special cases that often cause the most trouble and require the most time. A specification generated through the reverse translation algorithm makes explicit to the user many of these special cases by showing all conditions and naming each value explicitly through the single value variables. The resulting specification also makes explicit the order of evaluation and sharing of data with other processes through its file declarations.

In comprehending the specification it is important to remember that the specification is NOT a procedural language program. A user should not attempt to comprehend the specification by asking, "what is the specification doing?" Rather, the user should view the specification as a set of equations that define values and ask what are the values being defined.

```
PROCEDURE: Enter (i) EXTERNAL(sharedin1,sharedin2,sharedin3,sharedout);

SOURCE: i, sharedin1, sharedin2, sharedin3;
TARGET: sharedout;

i IS FIELD (DEC);

1 sharedin1 IS FILE ORG=SHARED,
  2 c_7 IS FIELD (BIT(1)) RENAME AS c[1:2],
  2 grp1(0:1) IS GROUP,
    3 grp2(*) IS GROUP,
      4 turn_8 IS FIELD (DEC) RENAME AS turn,
      4 c_14(1:2) IS FIELD (BIT(1)) RENAME AS c[1:2];

1 sharedin2 IS FILE ORG=SHARED,
  2 grp1(0:1) IS GROUP,
    3 grp2(*) IS GROUP,
      4 turn_10 IS FIELD (DEC) RENAME AS turn,

1 sharedin3 IS FILE ORG=SHARED,
  2 grp1(0:1) IS GROUP,
    3 grp2(*) IS GROUP,
      4 grp3(*) IS GROUP,
        5 turn_11 IS FIELD (DEC) RENAME AS turn;

1 sharedout IS FILE ORG=SHARED,
  2 c_6 IS FIELD (BIT(1)) RENAME AS c[1:2],
  2 grp1(0:1) IS GROUP,
    3 grp2(*) IS GROUP,
      4 c_9(1:2) IS FIELD (BIT(1)) RENAME AS c[1:2],
      4 c_12(1:2) IS FIELD (BIT(1)) RENAME AS c[1:2],

c_7(j_5) = DEPENDS_ON(c_6(i));

c_9(sub1,sl8(sub1,sub2),i) = DEPENDS_ON(turn_8(sub1,sub2));

turn_10(sub1,sl8(sub1,sub2)) = DEPENDS_ON(c_9(sub1,sl8(sub1,sub2),i);

turn\_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3) = DEPENDS_ON(turn\_10(sub1,sl8(sub1,sub2)));

c_12(sub1,sl8(sub1,sub2),i) = DEPENDS_ON(turn_11(sub1,sl8(sub1,sub2),sub3));

c_14(sub1,sub2,j_5) = DEPENDS_ON(c_12(sub1,sl8(sub1,sub2),i);
```

Figure 22: Specification for Enter - Header and Declarations

```
j_5 = 3-i;

c_6(i) = true;

c_9(sub1,sl8(sub1,sub2),i) = false;

c_12(sub1,sl8(sub1,sub2),i) = true;


slc8(sub1,sub2) = turn_8(sub1,sub2) = j_5;

sl8(sub1,sub2) = sublinear(slc8(sub1,sub2),sl8(sub1,sub2),sub2);

slc10(sub1,sl8(sub1,sub2)) = turn_10(sub1,sl8(sub1,sub2)) = j_5;

sl10(sub1,sl8(sub1,sub2)) = sublinear(slc10(sub1,sl8(sub1,sub2)),
                                      sl10(sub1,sl8(sub1,sub2)),sl8(sub1,sub2));


SIZE.slc8(*,*) = if c_7(j_5) then 1 else 0;

END.turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3) =
                        ^(turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3) = j_5);

END.slc8(sub1,sub2) = ^(c_14(sub1,sub2,j_5));
```

Figure 23: Specification for Enter - Equations

```
PROCEDURE: Leave (i) EXTERNAL(sharedout);

SOURCE: i;
TARGET: sharedout;

i IS FIELD (DEC);

1 sharedout IS FILE ORG=SHARED,
  2 turn_22 IS FIELD (DEC) RENAME AS turn,
  2 c_23(1:2) IS FIELD (BIT(1)) RENAME AS c[1:2];

turn_22 = 3-i;

c_23(i) = false;
```

Figure 24: Specification for Leave

6 Proving Correctness of The Specification

The key objective of reverse software engineering is to improve the understandability of programs. Our belief is if a program is easier to prove correct then it is also easier to understand. Therefore, we use ease of proving correctness as an indicator for ease of understandability.

Using the ease of proving correctness as an indicator, we claim the generated specification is easier to understand then the source procedural program. The following proof is presented in support of this claim.

In determining the correctness of Dekker's Algorithm it suffices to show the correctness of the specification generated by Reverse Software Engineering. There are three requirements that we must verify. They are: (1) mutual exclusion, (2) deadlock free and (3) bounded-wait.

Before preceding, we make the following observation. From Figure 8 it can be seen that process *pi* may only enter its critical section if Enter(i) has terminated. Therefore, proving the specifications satisfy the above three requirements is really proving appropriate termination of Enter(). The termination of Enter() is determined by the control variables SIZE.slc8(*,*), END.slc8(sub1,sub2) and END.turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3); they define the range of the first, second and third dimensions, respectively, of multi-dimensional variables. If SIZE.slc8(*,*)=0, the range of the first dimension is zero which by definition implies the other two variables do not exist, i.e. they are NULL. END.slc8(sub1,sub2) is TRUE for the last value in the second dimension. Likewise, END.turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3) is TRUE for the last value of the third dimension. Thus the specification terminates if SIZE.slc8(*,*)=0 or END.slc8(sub1,sub2)=TRUE and END.turn_11 is NULL or END.slc8(sub1,sub2)=TRUE and END.turn_11(sub1,sl10(sub1,sl8(sub1,sub2)),sub3)=TRUE.

The reader should keep in mind during the proofs the renaming of variables defined in

the specifications. Because the two processes may be executing both Enter(1) and Enter(2) concurrently, the same variable name may appear in a proof twice. Once for the occurrence in the specification executing in p1 and once for p2. To distinguish between the two referenced variable name, we prefix the name of the variable with the appropriate process name.

The approach we use for proving the three requirements is as follows. We first derive for each requirement the applicable situations. Then for each applicable situation, we generate a scenario of source variable value from which target values are determined. The values are then used as the base of our proof. Note that the process of generating scenarios may lead to the awareness of additional situations. In this way the two steps are performed iteratively.

Table 4 shows the applicable situations for each of the three proof requirements and the corresponding scenario. The applicable situations are derived in a goal/subgoal tree structure fashion. Table 5 shows the four generated scenarios. A scenario represents a computation of the specification with a given set of input values. This set of input values may constrain other input values.

Each situation in Table 4 defines initial values for the specification. With these initial values the target values defined by the specification are determined. For example, the first applicable situation for mutual exclusion states *pi* request entry into cs_i, *pj* is in cs_j and initially turn=i. These initial conditions are all that is necessary to determine the values of each input variable in the specification. They are shown in Table 5 in the column labeled scenario 2. The values of c_7(j_5) and c_14(1,sub2,j_5) are determined by the condition *pj* is in cs_j. turn_8(1,1) and turn_8(1,sub2) are determined by the initial condition turn=i. This is an example of one input value what another input value may be. A quick scan of the specification reveals that only *pi* changes turn to j and this occurs in specification Leave(i). Thus initial turn=i implies turn=i for all values of turn in specification Enter(i). With all input values determined, the values of all other variables are defined as shown in scenario 2.

Scenarios are similarly generated for all applicable situations. Our 12 situations result

in four scenarios. As might be expected more then one set of initial conditions may generate the same scenario. It may interest the reader that these four scenarios are also the only possible scenarios for Enter(i) and express all possible values of each variable.

The generation of scenarios is similiar to tracing a program with a set of initial input values. The difference is that all values are determined and summarized for immediate reference. Furthermore, the process of generating a scenario of values can be automated. All that is required is some set of initial source variable values. The automation of this process allows the analyst to interatively develop the applicable situations table with the aid of a computer. This can greatly reduces the possibility of incorrect results due to human error.

Requirements	Tree Structure of Applicable Situations	Scenario
Mutual Exclusion	*pi* request entry into cs_i and *pj* is in cs_j	
	initially turn=i	2
	initially turn=j	4
Progress	Both *pi* and *pj* request entry into their cs	
	initially turn=i	2
	initially turn=j	4
Bounded-wait	*pi* request entry into cs_i and *pj* not in cs_j	
	initially turn=i	1
	initially turn=j	1
	pi request entry into cs_i and *pj* in cs_j	
	pj leaves and does not want to re-enter cs_j	
	initially turn=i	2
	initially turn=j	
	†turn=i	3
	‡turn=i	4
	pj leaves and wants to re-enter cs_j	
	initially turn=i	2
	initially turn=j	
	†turn=i	3
	‡turn=i	4

†*pi*.turn_8(1,1)=j and *pi*.turn_10(1,1)=i follows *pj*.turn_23=i
‡*pi*.turn_11(1,1,sub3)=j and sub3=q follows *pj*.turn_23=i.

Table 4: Summary of Applicable Situations for the Three Requirements

	Variables	Scenario 1	Scenario 2	Scenario 3	Scenario 4
SOURCE	c_7(j_5)	FALSE	TRUE	TRUE	TRUE
	turn_8(1,1)	NULL	i	j_5	j_5
	turn_8(1,sub2)[1]	NULL	i	i	i
	turn_10(1,1)	NULL	NULL	i	j_5
	turn_11(1,1,sub3)[2]	NULL	NULL	NULL	j_5
	turn_11(1,1,q+1)[2]	NULL	NULL	NULL	i
	c_14(1,sub2,j_5)[3]	NULL	TRUE	TRUE	TRUE
	c_14(1,n+1,j_5) [3]	NULL	FALSE	FALSE	FALSE
TARGET	c_6(i)	TRUE	TRUE	TRUE	TRUE
	c_9(1,1,i)	NULL	NULL	FALSE	FALSE
	c_12(1,1,i)[4]	NULL	NULL	TRUE	TRUE
	slc8(1,1)	NULL	FALSE	TRUE	TRUE
	sl8(1,1)	NULL	0	1	1
	slc10(1,1)	NULL	NULL	FALSE	TRUE
	sl10(1,1)	NULL	NULL	0	1
Data Control Variables	SIZE.slc8(*.*)	0	1	1	1
	END.turn_11(1,1,sub3)[2]	NULL	NULL	NULL	FALSE
	END.turn_11(1,1,q+1)[2]	NULL	NULL	NULL	TRUE
	END.slc8(1,sub2)[3]	NULL	FALSE	FALSE	FALSE
	END.slc8(1,n+1)[3]	NULL	TRUE	TRUE	TRUE

[1]For all sub2>1

[2]For all sub3≤q, where 0≤q corresponds to *pj* defining *pj*.turn_22=i. If q=0, then sub3=0 and the variable is NULL.

[3]For all sub2≤n, where 0≤n corresponds to *pj* defining either *pj*.c_9(j)=FALSE or *pj*.c_23(j)=FALSE. If n=0, then sub2=0 and the variable is NULL.

[4]c_12(1,1,i) is defined only if END.turn_11 is NULL or END.turn_11(1,1,q+1)=TRUE.

Table 5: Scenarios Table

Mutual exclusion The requirement for mutual exclusion is that if one process (pj) is executing in its critical section (cs_j), the other process (pi) cannot be executing in its critical section (cs_i).

Proof. We have as a given condition process pj in cs_j and pi request entry into cs_i. According to Table 4 scenarios 2 and 4 summarize this situation. Scenario 2 shows the value of each variable in Enter(i) given this condition and turn=i. Scenario 4 show the value of each variable in Enter(i) given this condition and turn=j. From scenarios 2 and 3 it can be seen that pi.END.slc8(1,sub2)=FALSE for sub2$\leq$n, where n corresponds to either pj.c_9(j)=FALSE or pj.c_23(j)=FALSE. Thus we conclude pi does not enter cs_i as long as pj is in cs_j and mutual exclusion is observed. □

Progress If both processes wants to enter its cs simultaneously, one will be allowed to enter in a finite amount of time.

Proof. We have as our initial condition processes pi and pj both request entry simultaneously. Without loss of generality, we assume turn=i.

From Table 4 we know that Scenario 4 shows the value of each variable in Enter(j) for this initial condition. According to Scenario 4, pj.END.turn_11(1,1,sub3)=FALSE for sub3$\leq$q, where q corresponds to pi defining pi.turn_22=j. Thus pj does not enter cs_j.

Scenario 2 shows the value of each variable in Enter(i) for the same initial condition. Scenario 4 shows pi.END.turn_11 is NULL and pi.END.slc8(1,n+1)=TRUE, where n corresponds to pj defining pj.c_9(1,1,j)=FALSE. Thus pi enters cs_i for some value n and progress is observed. □

Bounded-wait The requirement for bounded-wait is that there is a bound in the number of times process pj may enter cs_j between the moment process pi makes a request to enter cs_i and the moment the request is granted.

Proof.

Case 1. pi request entry into cs_i and pj is not in, nor requesting to enter cs_j.

Scenario 1 summarizes this situation. But Scenario 1 shows END.turn_11 and END.slc8 are NULL. Thus pi enters cs_i immediately.

Case 2. pi request entry into cs_i and pj is in cs_j leaves and does not want to re-enter cs_j.

Subcase 1. Initially turn=i.

This condition is summarized by Scenario 2. Scenario 2 shows pi.END.turn_11 is NULL and pi.END.slc8(1,n+1)=TRUE, where n corresponds to pj defining pj.c_23(j)=FALSE via Leave(j). Thus pi enters cs_i for some value n.

Subcase 2. Initially turn=j.

This condition is summarized by scenarios 3 and 4. Scenario 3 shows pi.END.turn_11 is NULL and pi.END.slc8(1,n+1)=TRUE for some n corresponding to pj defining pj.c_23(j)=FALSE via Leave(j). Thus pi enters cs_i for some value n.

Scenario 4 shows pi.END.turn_11(1,1,q+1)=TRUE for some q corresponding to pj defining pj.turn_22=i, and pi.END.slc8(1,n+1)=TRUE for some n corresponding to pj defining pj.c_23(j)=FALSE. Thus pi enters cs_i for some values q and n.

Case 3. pi request entry into cs_i and pj in cs_j leaves and wants to re-enter cs_j.

Subcase 1. Initially turn=i.

If pi detects pj has left cs_j, i.e. pi.c_14(1,n+1,j)=FALSE, then this is the same as Subcase 1 of Case 2.

On the other hand, if pj is sufficiently fast to reference Enter(j) before pi detects pj has left cs_j then we have both pi and pj requesting entry into their cs and turn=i. But this is equivalent to proving progress where we have already shown pi enters cs_i.

Subcase 2. Initially turn=j.

This condition again corresponds to scenario 3 and 4. If pi detects pj has left cs_j, i.e. pi.c_14(1,n+1,j)=FALSE, then this is the same as Subcase 1 of Case 2. So we assume that pj is sufficiently fast to reference Enter(j) before pi detects pj has left cs_j.

Scenario 3 shows pi.c_9(1,i)=FALSE. Thus pj may re-enter its cs any number of times. But Scenario 3 also shows pi.END.turn_11 is NULL and pi.c_12(1,i)=TRUE. Thus there is a bound on the number of times pj may re-enter its cs. So for some instance we have both pi and pj requesting to enter their respective cs and turn=i. But this is equivalent to showing progress where we have shown pi enters cs_i.

Scenario 4 shows pi.c_9(1,i)=FALSE. Again pj may re-enter its cs any number of times. But Scenario 4 shows pi.END.turn_11(1,1,q+1)=TRUE for some q corresponding to pj defining pj.turn_22=i and pi.c_12(1,i)=TRUE. Thus there is a bound on the number of times pj may re-enter its cs. So for some instance we have both pi and pj requesting to enter their respective cs and turn=i. This is again equivalent to showing progress where it was shown pi enters cs_i.

Thus for all possible cases pi enters cs_i and the requirement of bounded-wait is observed. □

To summarize, we proceed a proof by the derivation of the applicable situations and the generation of scenarios. We then perform the proof by referencing the values defined in the scenarios. In contrast, a proof given in [3] involves tracing the execution of the procedural program.

7 SURVEY OF TEMPORAL LOGIC

"Temporal logic is a logic of propositions whose truth and falsity may depend on time" [5]. Its interest lie in its usefulness in analysis and formalization of concurrent programs.

An execution of a program can be viewed as a sequence of states, called execution states, that undergo a series of transformations determined by the program's instructions [8]. In different states, program entities such as variables may have different values. Temporal logic enables us to discuss this change of situation over time in a formal setting.

We make no attempt here to fully formalize temporal logic. Our only interest is to present an example of how a proof in temporal logic may proceed and contrast this with the alternative approach given in Section 6. Any reader not familiar with temporal logic and wishing more detail is encouraged to read [5, 8, 9].

The execution of a concurrent program may be modeled as taking place in discrete steps. That is, although two or more processes of a program may be executing in parallel, only one statement from all processes is executed in a given step. Thus an execution of a concurrent program can be viewed as a sequentialization of its statements. The order of execution, however, is arbitrary and may vary on each run. This model of computation is called the *interleaving model* and the justification of this model is given in [5]. It is on the basis of this model that we give our example proof.

Before we present our example proof in temporal logic we need to first lay a foundation on which to proceed. We assume some minimal familiarity with temporal logic on the part of the reader.

The following definitions and rule were obtained from [5]. The invariant rule we give is stated without proof as the proof is already given in [5]. This is justified since reproduction of the proof here would mean the formalization of temporal logic which we have already stated is not our goal.

Logical operators and their meanings:

$\circ A$: "A holds at the time point immediately after the reference point" (*nextime* operator).

$\Box A$: "A holds at all time points after the reference point" (*always* operator).

$\Diamond A$: "There is a time point after the reference point at which A holds" (*eventually* operator).

Other meanings:

α "The action represented by α is executed (next)".

at α "The action represented by α is the next one to be executed", ("α is ready to execute").

at c_i "One of the action represented by a member of set c_i is the next one to be executed."

Abbreviations:

C **invof** α for $\alpha \wedge C \rightarrow \circ C$

("C is an invariant of α"),

C **invof** M for C **invof** $\alpha_1 \wedge \ldots \wedge C$ **invof** α_n, where $M=\{\alpha_1,\ldots,\alpha_n\}$

("C is an invariant of every $\alpha \in M$").

exor $(A_1, ..., A_k)$ for $(A_1 \wedge \neg A_2 \wedge ... \wedge \neg A_k) \vee$
$(\neg A_1 \wedge A_2 \wedge ... \wedge \neg A_k) \vee$
$\vdots$
$(\neg A_1 \wedge ... \wedge \neg A_{k-1} \wedge A_k) \vee$

("Exactly one of $A_1, ..., A_k$ is true"),

excl $(A_1, ..., A_k)$ for **exor** $(A_1, ..., A_k) \vee (\neg A_1 \wedge ... \wedge \neg A_k)$

("At most one of $A_1, ..., A_k$ is true").

Invariance rule:

(inv) $A \rightarrow B$, B **invof** $M_\Pi \vdash \mathrm{A} \rightarrow \Box B$.

This rule states "in order to prove that B holds in every execution state from a state in which A holds, show that B is true in this first state and is invariant under every action" [5].

A label, say α_i is associated with each statement of the program. This label is unique and does not occur as a label for any other statement in the program.

Proving Mutual Exclusion To contrast the difference between the approach given in Section 6 to that of temporal logic, we will again show mutual exclusion for Dekker's algorithm. The proof is performed with respect to the original procedural language program of Figure 7.

As stated previously, we must show that if one process (pj) is executing in its critical section (cs_j), the other process (pi) cannot be executing in its critical section (cs_i).

We perform the following proof by the invariant method. In this method, we merely state mutual exclusion as an assertion and show this assertion is invariant under every action of the program.

For brevity let Π represent the main program, i.e. the program the two processes reside in. We then observe the following facts about Π:

F1: For pi to enter cs_i, it must be that c[j]=FALSE.

F2: pi sets c[i]=TRUE before entering cs_i.

F3: pi always sets c[i]=FALSE after leaving cs_i.

F4: c[i]=TRUE while pi is in cs_i.

F5: Only pi changes the value of c[i].

Proof. Let α_k be the label for line k in Enter(i) and Leave(i) of Figure 7. Likewise, let β_k be the label for line k in Enter(j) and Leave(j). Finally, let c_i and c_j be the sets of labels in the critical sections of *pi* and *pj*, respectively. Then the assertion of mutual exclusion is:

$$\Pi \vdash Start_\Pi \rightarrow \Box \textbf{excl}\ (\text{at } c_i, \text{at } c_j).$$

To show mutual exclusion we need only show that the assertion of mutual exclusion is derivable.

Derivation of the assertion. Let $A \equiv \textbf{excl}\ (\text{at } c_i, \text{at } c_j)$ and let M_Π be the set of all labels in Π. The derivation is as follows:

(1)	$Start_\Pi \rightarrow A$	definition of start
(2)	A **invof** $M_\Pi \backslash \{\alpha_6,...,\alpha_{14},\alpha_{22},\alpha_{23},\beta_6,...,\beta_{14},\beta_{22},\beta_{23}\}$	since without occurrence of c[i],c[j] and turn
(3)	at $\{\alpha_6, ..., \alpha_{14}, \alpha_{22}, \alpha_{23}\} \wedge A \rightarrow \neg$at c_i	Π
(4)	$\{\alpha_6, ..., \alpha_{13}, \alpha_{22}, \alpha_{23}\} \wedge A \rightarrow \circ A$	Π, (3)
(5)	$\alpha_6 \rightarrow \circ$c[i]=TRUE	Π
(6)	$\alpha_9 \rightarrow \circ$c[i]=FALSE	Π
(7)	$\alpha_{12} \rightarrow \circ$c[i]=TRUE	Π
(8)	$\alpha_9 \rightarrow \Diamond\alpha_{12}$	Π
(9)	at $\alpha_{14} \rightarrow$ c[i]=TRUE $\wedge$ c[j]=FALSE	Π,(5),(6),(7),(8)
(10)	c[i]=TRUE $\wedge$ c[j]=FALSE $\rightarrow \neg$at c_j	Π,F1,F2,F3,F4
(11)	at $\alpha_{14} \wedge A \rightarrow \neg$at $c_i \wedge \neg$at c_j	Π,(9),(10)
(12)	$\alpha_{14} \wedge A \rightarrow \circ A$	Π,(11)
(13)	A **invof** $\{\beta_6,...,\beta_{14},\beta_{22},\beta_{23}\}$	in the same way as (4) and (12)
(14)	A **invof** M_Π	(2),(4),(12),(13)
(15)	$Start_\Pi \rightarrow \Box A$	(inv), (1), (14) □

We point out that this is not a derivation in the strict sense. Some steps have been abbreviated and purely logical rules and laws have been omitted. This was done to minimize time spent on developing temporal logic for this example.

This proof technique is mathematically rigorous and is widely regarded as a promising approach to proving program correctness. However, this approach requires the user to be

knowledgeable in (temporal) logic.

The approach we have presented in Section 6 requires no understanding of any formal logic. All that is required is some basic understanding of regular and boolean algebra.

The process of proving an assertion in temporal logic is entirely manual. The analyst must first formally state the assertion he/she wishes to prove. Then he/she must manually step through the program with no aid.

On the otherhand, the approach in Section 6 is largely automated. Information is gained both interactively and automatically. The applicable situations are derived while interacting with the computer. The computer also automatically generates the scenario of variable values determined by the initial source variable values. No manual stepping through of the specification is require.

8 CONCLUSION

Reverse Software Engineering is a method to utilize outdated programs to reduce the cost of developing new replacement systems. The old computer systems are assumed to be inadequate in functionality and implementation technology. Still, to reduce cost it is desired to reuse what is available in the old system as a basis for making appropriate modifications. This is becoming increasingly important as the rapid advancements in new technology quickly age existing systems [11].

One of the major problems with software development and modification is the need of the user to "think like a computer." By translating the procedural language program into a specification we relieve the user of computer oriented concerns such as execution sequences, loops, side effects and efficiency. The generated specification contain only dataflow information and rules for data transformation [6]. All analysis and modification are performed on the specification. Efficient programs are then automatically generated from the specification.

By using a mathematical representation as a medium for understanding, analyzing and modifying old programs we can avoid many of the difficulties inherent in procedural language programs and reduce software development costs. This is the underlying notion of this paper.

The major contribution of this paper is the algorithm for translating concurrent procedural programs into specifications, and especially the new treatment of shared memory. To maintain in the specification the time precedences in the program, declarations of the shared memory specify order of references and use a renaming scheme to map these declarations to the structure of the real shared memory.

Another contribution is the methodology of proving program correctness presented in Section 6. By using the Scenario Table the reader can more easily comprehend the values being defined and prove assertions about the specification.

We have also described graphical ways of presenting information about the specification, the generation of which may be automated. This is the dataflow diagram of shared memory, the Applicable Situations table and the Scenarios table.

References

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

[2] Computer Command and Control Company, *The MODEL Language Usage and Reference Guide - Non-Procedural Programming for Non-Programmers*, 2300 Chestnut St., Philadelphia, PA 19103, 1987.

[3] E. Dijkstra, "Co-operating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965; reprinted in [4], pages 43-112.

[4] F. Genuys, (Editor), *Programming Languages*, Academic Press, London, 1968.

[5] F. Kroger, "Temporal Logic of Programs," EATCS Monographs on Theoretical Computer Science, Vol. 8, 1987.

[6] L. Liu, "Specification-Based Interface Checking of Large Real-Time/Distributed Systems," *Ph.D. Thesis*, University of Pennsylvania, July 1988.

[7] M.Maekawa, A. Oldehoeft and R. Oldehoeft, "Operating Systems Advanced Concepts," The Benjamin/Cummings Publishing Company, Inc., 1987.

[8] Z. Manna and A. Pnueli, "Verification of Concurrent Programs, Part I: The Temporal Framework," Research sponsored by the Office of Naval Research, Report No. STAN-CS-81-836, Stanford University, June 1981.

[9] Z. Manna and A. Pnueli, "Verification of Concurrent Programs, Part II: Temporal Proof Principles," Research sponsored by the Office of Naval Research, Report No. STAN-CS-81-843, Stanford University, September 1981.

[10] D. Parnas, "Software Aspects of Strategic Defense Systems," American Scientist, Vol. 73, September-October, 1985, pp.432-440.

[11] N. Prywes, X. Ge, I. Lee and M. Song, "Reverse Software Engineering," Research sponsored by the Air Force Office of Scientific Research, Contract AFOSR-88-0116, University of Pennsylvania, December 1989.

[12] A. Silberschatez and J. Peterson, *Operating Systems Concepts*, Addison-Wesley Publishing Company, 1988.