2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

2

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED FINAL/01 AUG 90 TO 31 AUG 93

4. TITLE AND SUBTITLE

AN ENVIRONMENT FOR VISUALIZATION, RELIABILITY, & KNOWLEDGE ACQUISITION IN EQUATIONAL PROGRAMMING (U)

5. FUNDING NUMBERS

2304/A7
AFOSR-90-0335

6. AUTHOR(S)

Professor Noah Prywes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Pennsylvania
Dept of Computer & Infor. Sci.
3401 Walnut Street, Room 234-A
Philadelphia, PA 19104

8. PERFORMING ORGANIZATION REPORT NUMBER

AFOSR-TR- 94 0255

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
110 DUNCAN AVE, SUITE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFOSR-90-0335

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

UL

13. ABSTRACT *(Maximum 200 words)*

We investigated the concept of a visual software environment which facilitates man-machine cooperation during software development. The focus is on "oracle" operations performed by a human user during the man-machine cooperation. In the environment, graphics and equations are combined to enhance software understanding that is essential in software development. The environment consists of the following components: (1) visual programming: an icon-based graph editor is used for composing an array graph of an equational language program, for interactive syntax analysis, and for consistency checking of the array graph and equations. (2) compilation: an equational language program is statically checked in accordance with its semantics during compilation. (3) equational visual testing: test adequacy criteria are defined for the equational visual testing; the testing process becomes simple and intuitive; oracle operations such as path selection, path examination, finding test input values, monitoring execution, and evaluation are facilitated. (4) verification: equational reasoning is combined with graphical representation of programs. (5) knowledge acquisition: expertise in old legacy code in procedural languages such as algorithms and methods is transferred to rules of knowledge bases via equations.

14. SUBJECT TERMS

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR(SAME AS REPORT)

AN ENVIRONMENT FOR VISUALIZATION, RELIABILITY, AND KNOWLEDGE ACQUISITION IN EQUATIONAL PROGRAMMING

TECHNICAL REPORT

April 1993

Jee-In Kim

Department of Computer and Information Science

School of Engineering and Applied Science

University of Pennsylvania

Prepared Under Contract AFOSR-90-0335A

from the

Air Force Office of Scientific Research

Bolling Air-Force Base, DC 20332-6448

Abstract

We investigated the [redacted] concept of a visual software environment which facilitates man-machine cooperation during software development. The focus is on "oracle" operations performed by a human user during the man-machine cooperation. In the environment, graphics and equations are combined to enhance software understanding that is essential in software development.

The environment consists of the following components: (1) **visual programming**: an icon-based graph editor is used for composing an array graph of an equational language program, for interactive syntax analysis, and for consistency checking of the array graph and equations. (2) **compilation**: an equational language program is statically checked in accordance with its semantics during compilation. (3) **equational visual testing**: test adequacy criteria are defined for the equational visual testing; the testing process becomes simple and intuitive; oracle operations such as path selection, path examination, finding test input values, monitoring execution, and evaluation are facilitated. (4) **verification**: equational reasoning is combined with graphical representation of programs. (5) **knowledge acquisition**: expertise in old legacy code in procedural languages such as algorithms and methods is transferred to rules of knowledge bases via equations.

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Problem

The objective of this dissertation is to investigate concept of a software environment where a human user performs software development tasks such as composition, compilation, testing, verification, and knowledge acquisition using graphical tools. The crucial element of the environment is **man-machine cooperation** during developing software systems. **Software understanding** is essential to the man-machine cooperation. It is well-known that a graphical representation of software facilitates software understanding. The question is how to use graphics for facilitating the man-machine cooperation. The research is focused on human tasks, specially "oracle" operations, that are required to achieve the software development tasks. The research will show how graphs, tables, and software tools help human users in performing difficult software development tasks.

1.2 Contributions

Figure 1.1 illustrates the concept of software environment and man-machine cooperation. The graphical tools facilitate the software development tasks such as composition, compilation, testing, verification, and knowledge acquisition.

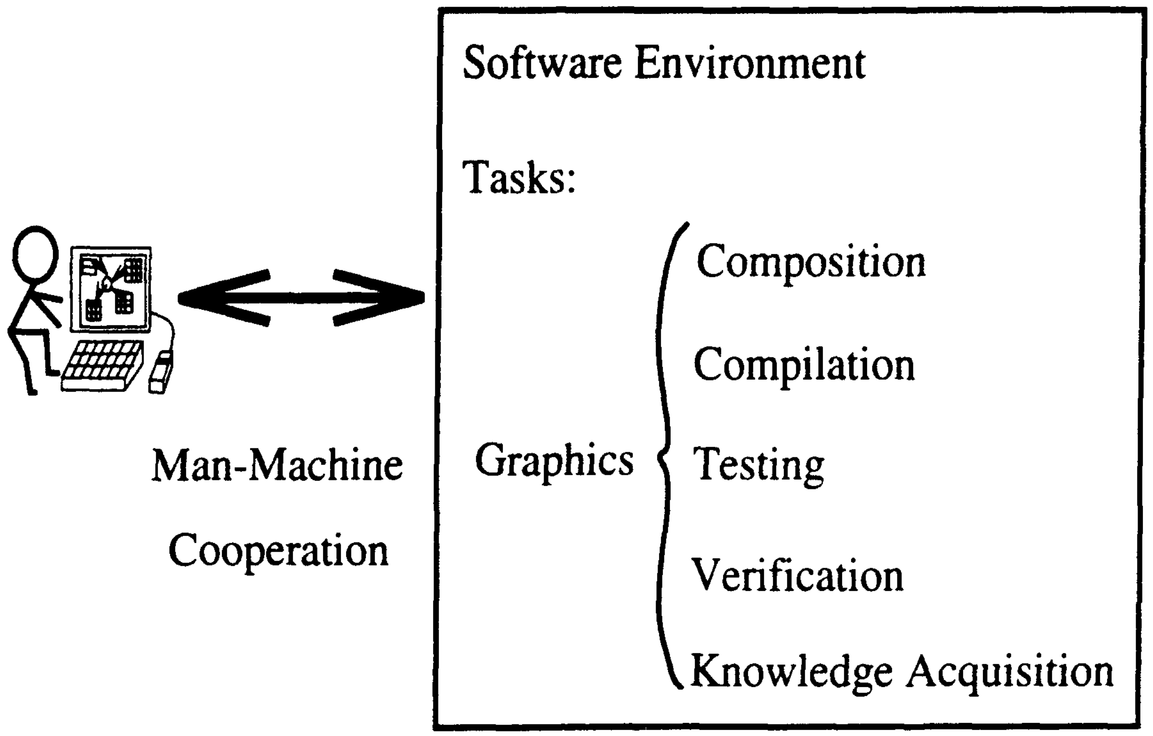


Figure 1.1: Software Environment and Man-Machine Cooperation.

A visual software environment is particularly effective when it is applied to equational programming. Figure 1.2 provides an overview of the visual software environment for equational programming. It provides interactive graphical tools for visual composition, compilation, testing, verification, and knowledge acquisition.

The environment enables programmers to perform equational programming visually via a data dependency graph called *array graph* [Lu81, PP83]. They draw array graphs and enter equations and declarations. The array graph facilitates understanding of inherent algorithms of programs by visualizing dependencies among equations and variables. It is difficult to describe such data dependencies in textual form. The visualized graph expresses the dependencies in terms of nodes and edges. It makes equational programming easy to learn and use.

Syntax analysis and compilation of the equational language are visually performed as a user composes an array graph which represents an equational language program. The environment also provides a new testing method, called *Equational*

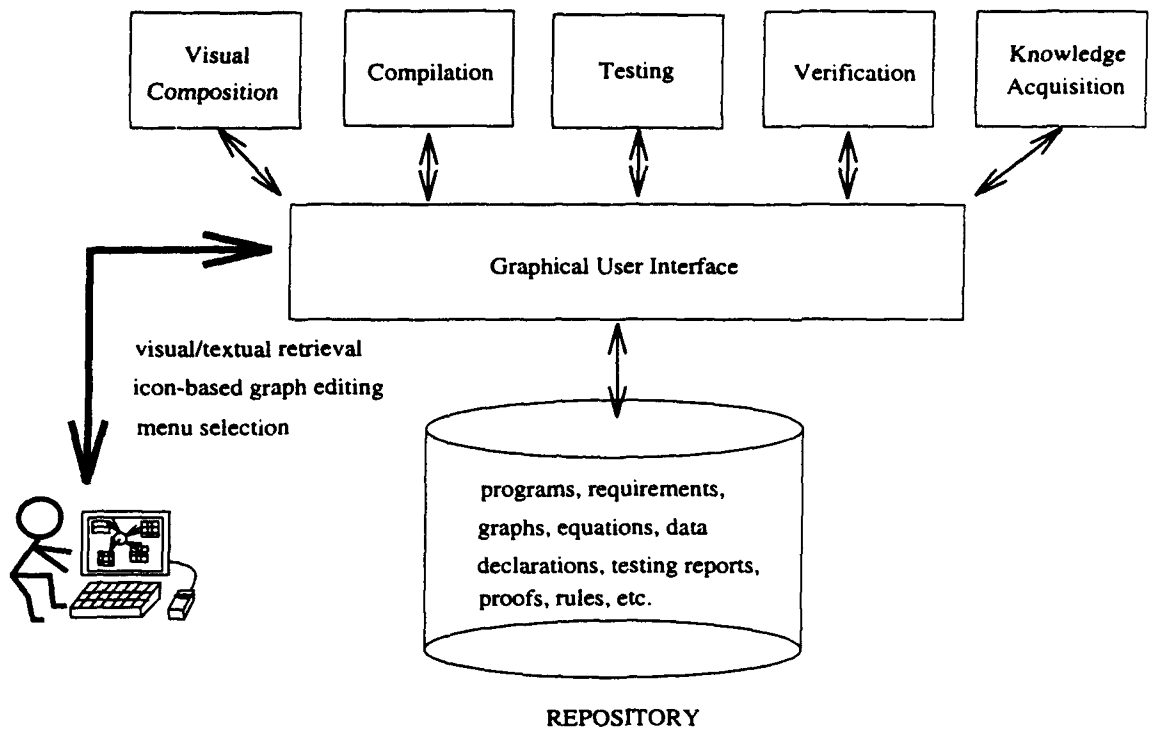


Figure 1.2: Visual Software Environment.

Visual Testing, based on analysis of array graphs. Equational reasoning is performed to construct proofs of assertions about the correctness of a program.

The graphical user interface facilitates man-machine cooperation. The repository of the environment contains programs, requirements, graphs, equations, testing reports, proofs, rules, etc.

The environment has innovative aspects in visual programming, compilation, testing, verification, and knowledge acquisition as follows:

(1) Visual Programming:

A user composes a program by drawing an array graph. An array graph helps a user in perceiving programs. Algebraic definitions of variables and equations are precisely expressed in an equational language. The visual programming method can express a complicated system using graphics and equations.

A node of an array graph denotes a variable or an equation. An edge visualizes data dependencies among equations and their associated variables, hierarchical structures of variable nodes, or parameter precedence between variables. Every variable is defined by a unique equation. Every equation node is fired as soon as all of its inputs are available. Therefore a user can examine a variable, an equation, or a group of connected variables and equations locally in the displayed array graph. A separate navigation window contains a map of a whole graph so that a user can easily traverse the graph. Syntax of an equational language [1] and an array graph [2] can be interactively checked as a user draws graphs. It is implemented by syntax-directed graph editing.

An interactive syntax analysis is performed visually, while an array graph is composed. The interactive syntax analysis makes the visual programming reliable by detecting errors at the early stage of program development.

(2) Compilation:

A composed array graph is checked with respect to the semantics of the equational language during compilation. The requirement of an equational language states that every variable must be defined. The equational compiler of the environment examines whether (1) there are ambiguous definitions and incomplete definitions of variables and equations in programs, (2) variables references are consistent in their dimensionalities, data types of variables and ranges of subscript expressions, (3) there is a causality path that computes a solution set for a given input values, and (4) conditions of terminating iterations are specified.

The equational compiler produces warning/error messages whenever inconsistency or incompleteness is found. It is not easy to interpret the messages and relate them to their associated nodes and edges on a visualized array graph. The environment displays the error/warning messages in a separate message window. When a user wants to examine a message, he clicks a corresponding

[1]See Appendix A.
[2]See Appendix B.

message icon on the message window. The environment locates nodes and/or edges on the array graph which are associated with the message. Then it graphically emphasizes them, i.e. shading and high-lighting. It facilitates a user's debugging during the visual programming.

(3) Testing:

A new testing method for equational programming (Equational Visual Testing) is based on the array graph. The graph directly visualizes data flow of an equational specification. The test adequacy criteria of the Equational Visual Testing are defined in terms of visualized paths of the array graph. Test adequacy criteria, similar to those used in testing of procedural programs, are: (1) The *equational all-paths* criterion defines a finite number of source-to-target paths based on acyclic array graphs. This makes the criterion practical. (2) The *equational all-du-paths* criterion defines a two-edge path, from an equation node, where a variable is defined, to equation nodes, where the variable is used. The path can be selected for traversal by providing test input values that satisfy at most two conditions. (3) The *equational all-uses* criterion is based on a single-edge path from a variable node to an equation node where the variable is used. The path can be traversed by satisfying at most one condition. This reduces labor of the testing in length of conjunction of the conditions. (4) The *equational all-definitions* criterion is shown to be trivial. Any single execution will traverse all definition paths.

The visual software environment facilitates performing the "oracle" operations in the testing as follows: (1) A path is traversed by satisfying conditions on the path. For each test adequacy criterion, a different set of paths are required to be traversed. Given test adequacy criterion, the environment generates a condition table where conditions and conjunctions of conditions are listed. The conditions in the table must be satisfied in order to satisfy the test adequacy criterion. (2) A human tester uses an array graph for selecting paths according to a specific test adequacy criterion. (3) Note that a path can be traversed by satisfying the conjunction of all conditions on the path. If any of the condition or the conjunction of the conditions is unsatisfiable, the path is not feasible. The human tester can use the visualized path and the condition table during the decision making. (4) The environment shows a causality path from the respective input variable nodes to a specific condition node in the array graph. It is used in backtracking input values. This reduces the labor of finding the test input values. (5) The human tester can view interim values of the variables and also refer to a specific iteration or a specific index of an array element. This is possible because every element of array variables is defined and the interim values are recorded during test execution. This further reduces the labor of evaluating test results. (6) The progress of testing is visually expressed via the array graph. It is useful during the evaluation.

(4) Verification:

An equational language program consists of equations. The correctness of the program is verified by checking if assertions about the correctness (also expressed as equations) can be derived from the equations of the program and general algebraic laws of arithmetic and logical operations using deduction rules. The basic deduction rules of equational reasoning are Reflexivity, Symmetry, Transitivity, Replacement, and Substitutivity. Induction, Case Analysis, and Tactic are added to the deduction rules. There is no need to trace program states during program verification because of the single assignment rule and the referential transparency. Note that program verification of the procedural paradigm is quite complicated because transition of program states must be examined. It requires a highly trained expert. A user guides the verification system in deducing equations from programs and general algebraic laws based on deduction rules. An inference engine handles the complexity of exercising program verification by mechanically applying the rules to deduce expressions.

It is critical to understand software in verification. The array graph can used as a visual aides to facilitate software understanding during verification.

(5) Knowledge Acquisition:

There is valuable expertise in legacy programs that can be automatically translated to rules in order to enrich knowledge bases of rule-based expert systems. The environment allows users to extract such expertise from [illegible] legacy programs and accumulate it as rules in knowledge bases. The [illegible] lation is exercised through the following steps: (1) use an existing method [Lu81, GP89] to translate procedural language programs into equational language programs (2) translate the equational language programs to rule-based language programs[Kim91]. Through the translation steps, the expertise in programs can be transferred to knowledge bases. The translation reduces human labor in collecting expertise for a rule-based expert system.

1.3 Outline of the Dissertation

Chapter 2 provides an introductory example and an overview of the environment. An icon-based graph editor of the visual programming under the environment is described in Chapter 3. Compilation is combined with graphics by visualizing warning/error messages as presented in Chapter 4. Chapter 5 discusses the new Equational Visual Testing methodology. An equational reasoning system and its application to program verification are described in Chapter 6. A method of extracting expertise from old legacy programs via language translation is presented in Chapter 7. The grammar of the equational language is formally expressed in Appendix A. An attribute grammar for visualizing an array graph is formulated in Appendix B. Appendix C lists error/warning messages produced by the MODEL compiler.

Chapter 2

Overview

2.1 Introduction

This chapter discusses an overview of the visual software environment. The requirements of the environment are **visualization**, **reliability**, and **knowledge acquisition**.

An introductory example is given in Section 2.2. The equational language, MODEL, is the basis of the environment. The syntax and semantics of the language are described in Section 2.3. The visual programming is implemented in terms of an icon-based graph editor in the environment. Section 2.4 explains the visual programming using an icon-based graph editor. Checking, testing, and program verification are performed to increase reliability of programs. Section 2.5 discusses checking. Section 2.6 describes testing. Program verification through equational reasoning is presented in Section 2.7. Section 2.8 discusses knowledge acquisition in the environment which is exercised via extraction of rules from existing procedural programs. The rule extraction is based on language translation from the procedural programs to rule-based programs via equations.

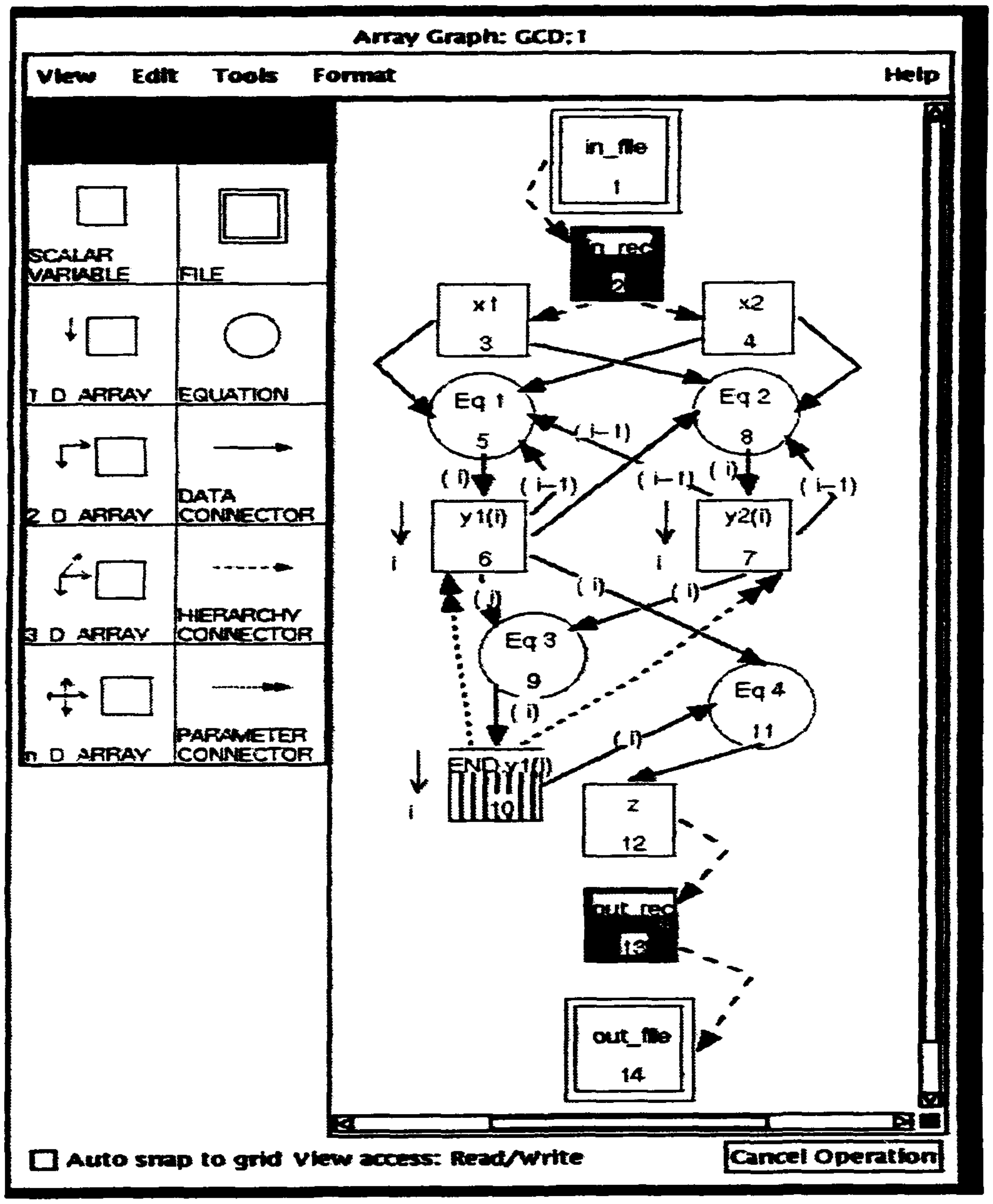


Figure 2.1: An array graph of Greatest Common Divisor.

2.2 Example

A user can develop a software system as follows: (1) The user draws nodes and edges of an array graph denoting an equational specification, (2) The user types data declarations and algebraic definitions of equations to complete the graph, (3) The user exercises syntax analysis and checking to find errors in the graph and the specification, (4) The user performs testing and program verification of the specification via the graph, and (5) The user translates the equations of the specification to rules for knowledge bases.

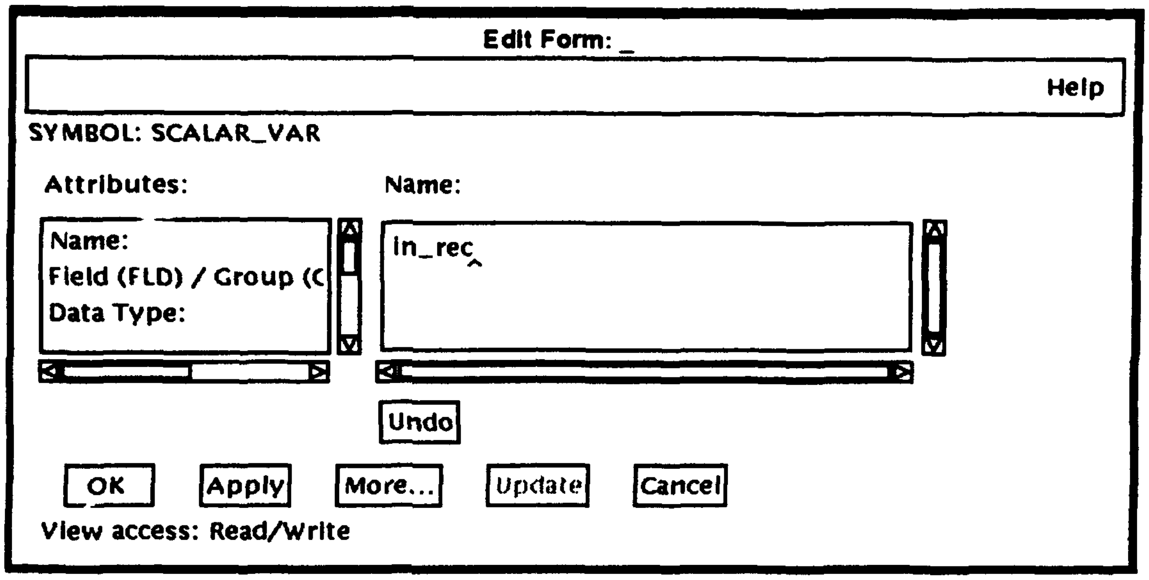
An icon-based graph editor for MODEL is illustrated in Figure 2.1. It displays an array graph which denotes a MODEL specification for finding the Greatest Common Divisor (GCD) based on the Euclid algorithm. The text corresponding to the graph is presented in Figure 2.2. Figure 2.1 shows a graphics window which consists of a canvas for drawing the graphs, a control panel containing icons of "SCALAR VARIABLE", "FILE", "1-D-ARRAY", "EQUATION", etc., and pull-down menus of "VIEW", "EDIT", "TOOLS", "FORMAT", and "HELP". The graph visually expresses the data declarations, the equations, and the dependencies between equations.

The *symbols* (nodes) of the array graph denote files, records, groups, variables, and equations. The *connectors* (edges) of the graph represent their data dependencies and hierarchical dependencies.

A user creates symbols by selecting icons from the control panel with a mouse. Either a data declaration or an algebraic definition of an equation must be entered for a symbol in textual form. The equational language, MODEL, is used in defining the data declaration and the equation. The text of the data declaration and the equation is entered by selecting a symbol and providing its definition into a separate form window. Figure 2.3 shows a form window for entering the data declaration of a record variable, `in_rec`. The symbol name is defined as "SCALAR_VAR" as displayed in Figure 2.3. The name of the symbol is entered as `in_rec` in the figure.

```
/* Header */
MODULE: GCD;
SOURCE: in_file;
TARGET: out_file;

/* Data Declarations */
1 in_file IS FILE,
  2 in_rec IS RECORD,
    3 x1 IS FIELD (PIC 'ZZZ9'),
    3 x2 IS FIELD (PIC 'ZZZ9');

1 out_file IS FILE,
  2 out_rec IS RECORD,
    3 z IS FIELD (PIC 'ZZZ9');

/* Subscripts */
(i) IS SUBSCRIPT;

/* Equations */

/* Eq 1 */
y1(i) = IF i=1 THEN IF x1>x2 THEN x1 ELSE x2
               ELSE IF y1(i-1)>y2(i-1) THEN y1(i-1) - y2(i-1)
                                       ELSE y1(i-1);
/* Eq 2 */
y2(i) = IF i=1 THEN IF x1>x2 THEN x2 ELSE x1
               ELSE IF y1(i-1)>y2(i-1) THEN y2(i-1)
                                       ELSE y2(i-1) - y1(i-1);
/* Eq 3 */
END.y1(i) = (y1(i) = y2(i));
/* Eq 4 */
z = IF END.y1(i) THEN y1(i);
```

Figure 2.2: A MODEL specification of Greatest Common Divisor in textual form.


Edit Form: _

SYMBOL: SCALAR_VAR

Attributes:	Name:
Name:	in_rec
Field (FLD) / Group (C	
Data Type:	

View access: Read/Write

Figure 2.3: Data declaration.

Then its sort, e.g., a field, a group, or a record, must be specified. In this particular example, it must be "RECORD". The data type should also be defined. An algebraic definition of an equation is entered via another form window. The syntax of the data declaration and the algebraic definition of the equation is checked with respect to the MODEL grammar presented in Appendix A.

A connector is created between two symbols to represent either data, hierarchical, or parameter dependency. A data connector is like the connectors (a solid arrow) between `x1` (node number 3) and `Eq 1` (node number 5) and `x2` (node number 4) and `Eq 1` in Figure 2.1. They denotes data dependency between variables and equations. That is, if the connectors comes from variable nodes to an equation node, it means that the variables are inputs of the equation; if the connector comes from an equation node to a variable node, it means that the variable is defined by the equation. A hierarchy connector like a connector (a broken arrow) between `in_file` (node number 1) and `in_rec` (node number 2) expresses a hierarchical dependency between the input file, `in_file`, and the record variable, `in_rec`. This means that the record variable is included in the input file. A parameter connector is denoted by a dotted arrow with double heads. The connector from `END.y1(i)` (node number

10) to y1(i) (node number 6) is a parameter connector. This means that the control variable, END.y1(i), represents the condition of determining the size of the array variable, y1(i).

Editing an array graph is completed when all of symbols are created, the symbols are connected by connectors, and the definitions of the symbols and the connectors are specified. Then the syntax and the semantics of the graph are examined. For example, a hierarchy connector can connect data symbols but it is not allowed to connect an equation symbol and a data symbol. The consistency between the graph and the texts (the data declarations and the algebraic definitions of the equations in MODEL) is also checked. The equation symbol, Eq 1 (node number 5 in Figure 2.1), is connected to an output data symbol, y1(i) (node number 6), and connected to four input data symbols, x1 (node number 3), x2 (node number 4), y1(i-1) (node number 6), and y2(i-1) (node number 7). Then the algebraic definition of the equation symbol must have a left-hand side (LHS) variable, y1(i), and right-hand side (RHS) variables, x1, x2, y1(i-1), and y2(i-1).[1] The definition must satisfy the MODEL syntax and semantics. The errors detected by the checking mechanism are reported in terms of warning/error messages in both textual form and graphical form.

There are test adequacy criteria defined for the Equational Visual Testing. The program is modified and compiled to be used in testing. The modification includes (1) expanding a complex conditional equation into a series of simple conditional equations (2) declaring interim variables as output variables to expose the interim values of the variables. An overview of the testing is discussed in Section 2.6. The more details is described in Chapter 5.

Equational reasoning is adopted for verification of the correctness of programs. First, proof goals are formulated in the form of a MODEL equation. Equations

[1]The labels of connectors represent subscript expressions of corresponding data dependencies. We know that y1(i-1) and y2(i-1) are inputs of Eq 1 without examing the algebraic definition of Eq 1. A user only defines the equation and the data declarations. Then the checking mechanism extracts subscript expressions for connectors and labels the connectors.

of a program to be verified are regarded as axioms during the reasoning. Then a proof or a disproof for each assertion is constructed by applying the deduction rules to the axioms and the assertion. The whole procedure of the program verification is dictated by a user via a graphical user interface. An overview of the program verification is presented in Section 2.7 and Chapter 6 discusses the more details.

Expertise encoded in algorithms and methods of programs can be translated to rules for knowledge bases. The procedure enriches the knowledge bases by inserting the rules. Section 2.8 discusses an overview and Chapter 7 presents the details.

2.3 Equational Language

MODEL is a high level mathematical language, i.e. an equational language such as EPL[Szy91], Haskell [Hud91], and MODEL [PP83, SP88, Hud89]. There are no implicit states, no side-effects and no predefined sequence of computation. It can be used for composing data declarations and algebraic definitions of equations that specify algorithms. A user can compose programs without considering implementation details [Hud89, Kim91]. A MODEL specification[2] consists of data declarations and equations. A MODEL specification can be understood by programmers with ease because it is based on regular and Boolean algebras. Formal program verification of the correctness of a MODEL specification is easier than that of procedural language programs, because it utilizes only algebraic manipulation of equations, namely equational reasoning, and there is no need to trace changes of program states. The reasoning is based on deduction rules by which equations can be deduced from the MODEL equations and general algebraic laws [Kim91].

A MODEL specification has the following requirements:

- A user must assure a causality (or a causal chain) for all the inputs of equations.

[2]The MODEL compiler translates MODEL code to a procedural language program such as C and Ada[Lu81, PLGS88]. Then the procedural language program is complied and executed. In order to distinguish the MODEL code from the procedural language program, it is called a MODEL specification.

- An equation must compute the unique value of its left-hand side (LHS) variable by evaluating its right-hand side (RHS) expression.

2.3.1 Functional Units

A functional unit in MODEL consists of a header, data declarations and equations. A header is an interface of a functional unit and specifies its type (module, function, or procedure), name and a list of inputs and outputs. A multi-unit specification consists of a main functional unit, called module and a set of subsidiary functional units, either functions or procedures [PLGS88]. A function accepts structures of input parameters and returns a single output structure. A procedure has input, output and update (treated as " new" and " old") parameters. As will be shown, definitions of subsidiary functions or procedures are in fact definitions of operations. An individual functional unit does not have recursive definition in itself, although it can use itself as an operation thus creating recursion. In the following, we focus on an individual module, function or procedure which are called "programs-in-the-small". For "programs-in-the-large", see [LP90].

2.3.2 Data Declaration

Data structures and their types are declared in a declaration part of a functional unit. There are input, output and interim variables. Input and output variables are declared in its header. A structure of each input and output variable, that is, an entire hierarchy of the structured variable, must be specified in a declaration part (it is keyed-in). Interim variables are used within a functional unit and cannot be accessed from the outside. Their declaration is optional. If there is no explicit declaration for an interim variable in a declaration part, a translator from MODEL to a procedural language inserts its declaration automatically. A primitive type of a variable is one of the followings: Boolean, integer, real, or literal. The primitive type is defined either explicitly or implicitly in the data declaration.

2.3.3 Equations

The syntax of a MODEL equation is formally defined in Backus-Naur Form (BNF) and presented in Appendix A. An equation is either a conditional equation or a simple equation. It defines an LHS variable in terms of an RHS expression. In composing a MODEL specification, only a variable, either a scalar variable or an array variable, is allowed in the LHS of the equality in an equation.[3] Array variables are indexed by subscript expressions.

An expression in the RHS of an equality defines the value of the LHS variable. Integer and real type variables are defined by arithmetic expressions. Boolean expressions define Boolean variables. Conditional expressions are built over either `IF-THEN` or `IF-THEN-ELSE`.

2.3.4 Array and Scalar Variables

A MODEL variable is either a scalar or an array. An array variable is indexed by a subscript expression. Just as in mathematics, each variable has a single value in MODEL. Once its value is assigned, the value never changes (the referential transparency). On the other hand, a subscript variable assumes all positive integer values in the range (size) of the elements of the arrays. Such subscripts are further discussed below.

Every array has a data declaration or an equation that defines its dimensionality, either implicitly or explicitly. For example, an equation `END.x(i) = exp(i)` may be defined for the range of an array **x** with a subscript `i`, where **exp** is a Boolean expression and a function of `i`. The variable `END.x(i)` is called a control variable [Lu81, MOD89]. **END** is prefixed to the array variable **x**. It is a "shadow" variable of **x** in the sense that it has the same shape as **x**. See Figure 2.4. The value (either **TRUE** or **FALSE**) of the control variable is defined by the equation `END.x(i) = exp(i)`.

[3]The restriction that only a variable is allowed in the LHS is relaxed by [Ge89]. In his extension, an LHS expression is defined as equal to an RHS expression. We use the extended MODEL language in formulating axioms and the MODEL calculus.

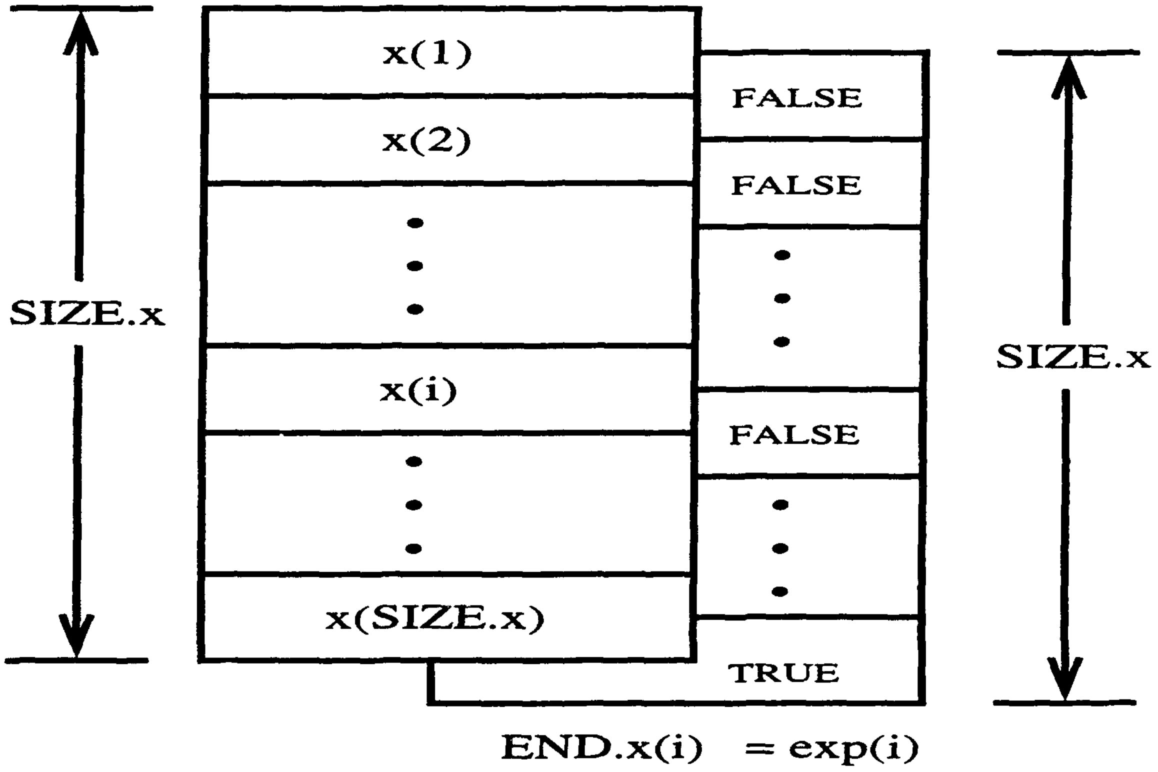


Figure 2.4: An array variable **x** and its "shadow" array variable `END.x`.

The values of `END.x(i)` are `FALSE` except for the value of the last element in the most right dimension that is `TRUE`. The size of the array variable `x` can be alternately defined directly by another prefixed control variable, `SIZE.x`. The array variable `x` is defined only when its subscript `i` satisfies a predicate $1 \leq i \leq$ `SIZE.x`. If the array is finite, `SIZE.x` has a finite value. Every element of `END.x` has the `FALSE` value while the last element `END.x(SIZE.x)` is `TRUE`. If `x` is an infinite array, however, there is no `TRUE` element in the array `END.x`, that is, the value of `SIZE.x` is infinite. It concludes that the following two equations are equivalent:

- **Rule of Control Variables:**

  `(END.x(i) = exp(i))` $\equiv$ `(exp(i) = IF (i = SIZE.x) THEN TRUE ELSE FALSE)`

2.3.5 Operations

The expressions of a MODEL specification use a set of operations. The operations are defined by operators, functions and their arguments. The followings are MODEL operators for expressions:

- Arithmetic Operators: $+$ (addition), $-$ (subtraction), $*$ (multiplication) and / (division).
- Relational Operators: $<$ (less than), $<=$ (less than equal), $>$ (greater than), $>=$ (greater than equal), $=$ (equal) and $! =$ (not equal).
- Logical Operator: & (and), | (or), ! (not), `IF-THEN`, and `IF-THEN-ELSE`.
- String Operator: || (concatenation), string search and string replacement.

Functions are viewed as operations on their input arguments. They are either built-in or user-defined.

2.3.6 Implicit Universal and Existential Quantifiers

The most distinctive difference between a procedural language and an equational language is that a variable has a single value in an equational language such as MODEL, namely the referential transparency. On the other hand, we can change the value of a variable in a procedural language as many times as we want to.

A MODEL equation, `x(i) = x(i-1) + 1`, defines all elements of `x` indexed by the subscript `i`. It means that the equation represents a class of all equations such that $\forall$ `i`, `1` $\leq$ `i` $\leq$ `SIZE.x`.

The MODEL equation, `x(i) = x(i-1) + 1`, can be interpreted into the following code in a procedural programming language such as FORTRAN:

```
DO I = 1, SIZEX
   X = X + 1
ENDDO
```

In FORTRAN, the assignment statement `X = X + 1` cannot be executed if the index variable `I` is out of its range, that is, either `I < 1` or `I > SIZEX`. It can be executed only when the index variable is properly defined, `1` $\leq$ `I` $\leq$ `SIZEX`. Likewise, a MODEL equation is defined only when its LHS variable subscript expression is within the range. If the LHS variable is size 0 or its subscript expression is out of range, *the equation is undefined. A MODEL equation that has undefined* LHS variable is invalid. Such an equation is called a null (or invalid) equation.

There is also the case of a null equation for specific subscript values: a conditional expression without `ELSE` in the RHS of the equation. Suppose we have the following equation: `y = IF i = 10 THEN x(i) + 1`. `y` is a scalar and its value is defined only for the equation instance of `i = 10`. The equation is invalid for other value of `i`. However there must be one instance (value of `i`) where the equation defines `y`, that is, `y` exists. In short, an instance of an equation becomes a null equation if its LHS variable is undefined or its RHS expression cannot define valid operations for its legal LHS variable.

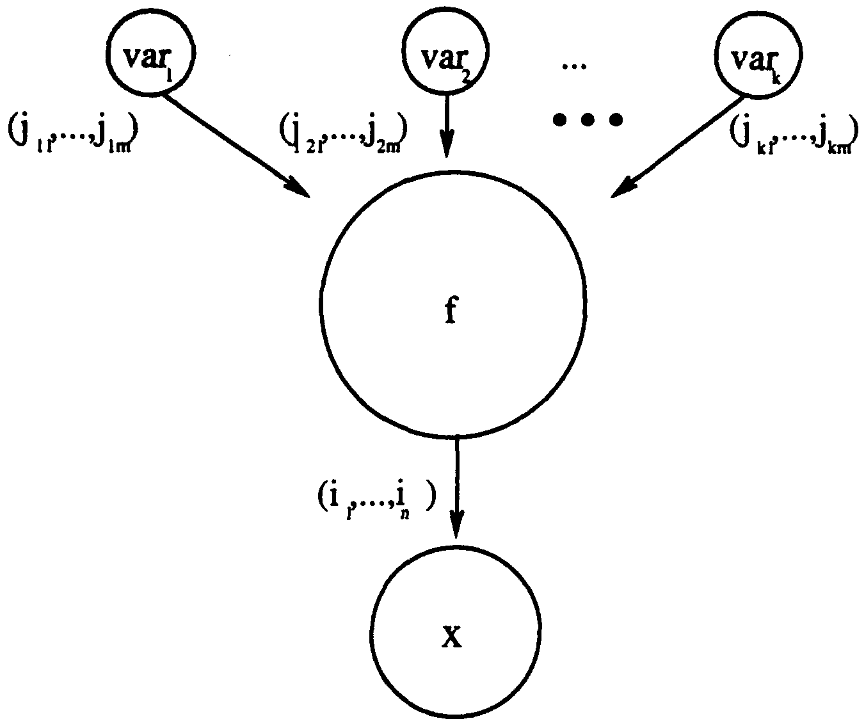


Figure 2.5: The Existence Condition.

A MODEL specification is regarded as a collection of valid equations. A valid equation is defined to be able to uniquely determine the value of its LHS variable in terms of its RHS expression. All the subscript expressions of every LHS variable should be defined within their legal range. There exists a unique RHS expression of a valid operation (that determines its value) for each element of an LHS variable. If multiple equations define the same LHS variable, they must be mutually exclusive of each other.

It concludes that existential (∃) and universal (∀) quantifiers implicitly exist for each MODEL equation. Consider an equation

$$\mathtt{x}(\mathtt{i}_1, ..., \mathtt{i}_n) = \mathtt{f}(\mathtt{j}_1, ..., \mathtt{j}_m, \mathtt{var}_1(\mathtt{j}_1, ..., \mathtt{j}_m), ...\mathtt{var}_k(\mathtt{j}_1, ..., \mathtt{j}_m))$$

where the LHS variable $\mathtt{x}$ is defined by the function $\mathtt{f}$; the LHS variable is indexed by the subscripts $\mathtt{i}_1, ..., \mathtt{i}_n$ and the function $\mathtt{f}$ may have subscripts $\mathtt{j}_1, ..., \mathtt{j}_m$ and the RHS variables $\mathtt{var}_1(\mathtt{j}_1, ..., \mathtt{j}_m), ..., \mathtt{var}_k(\mathtt{j}_1, ..., \mathtt{j}_m)$ as its arguments. If the ranges of the subscripts are $1 \leq i_1 \leq SIZE_{i_1}$, $1 \leq i_2 \leq SIZE_{i_2}$, etc., the equation can be interpreted as the following logical expression which denotes the existence condition of the variables and the equation:

$$\forall\, i_1, ..., i_n,\ 1 \leq i_1 \leq SIZE_{i_1}, ..., 1 \leq i_n \leq SIZE_{i_n},$$

$$\exists j_1, ..., j_m,\ 1 \leq j_1 \leq SIZE_{j_1}, ..., 1 \leq j_m \leq SIZE_{j_m},$$

$$\exists var_1(j_1, ..., j_m), ..., var_k(j_1, ..., j_m),$$

$$x(i_1, ..., i_n) = f(j_1, ..., j_m, var_1(j_1, ..., j_m), ..., var_k(j_1, ..., j_m))$$

The expression denotes the following:

> For all combinations of subscripts, $i_1, ..., i_n$, of the LHS variable, x, and the function, f, there exists at lease one combination of the subscripts in the legal range, $j_1, ..., j_m$, and all the RHS variables, $var_1(j_1, ..., j_m), ..., var_k(j_1, ..., j_m)$, of the function. The value of the LHS variable, x, is uniquely defined from the subscripts, the RHS variables and the function.

The existence condition of the equation, in fact, describes one of the requirements that must be satisfied by MODEL equations in a specification: Every variable must exist and be uniquely defined. The composition of such MODEL specifications is facilitated by the checking mechanism as will be discussed in Chapter 4. Incomplete, ambiguous and/or inconsistent definitions of subscript expressions, variables and equation are assumed to be removed after being detected by the checking mechanism. Any cyclic definition of variables is removed by the checking mechanism [Lu81, SLPP84].

2.4 Visual Programming

The environment offers an icon-based graph editor where a user can compose a MODEL specification by drawing an array graph on a graphics window and by typing

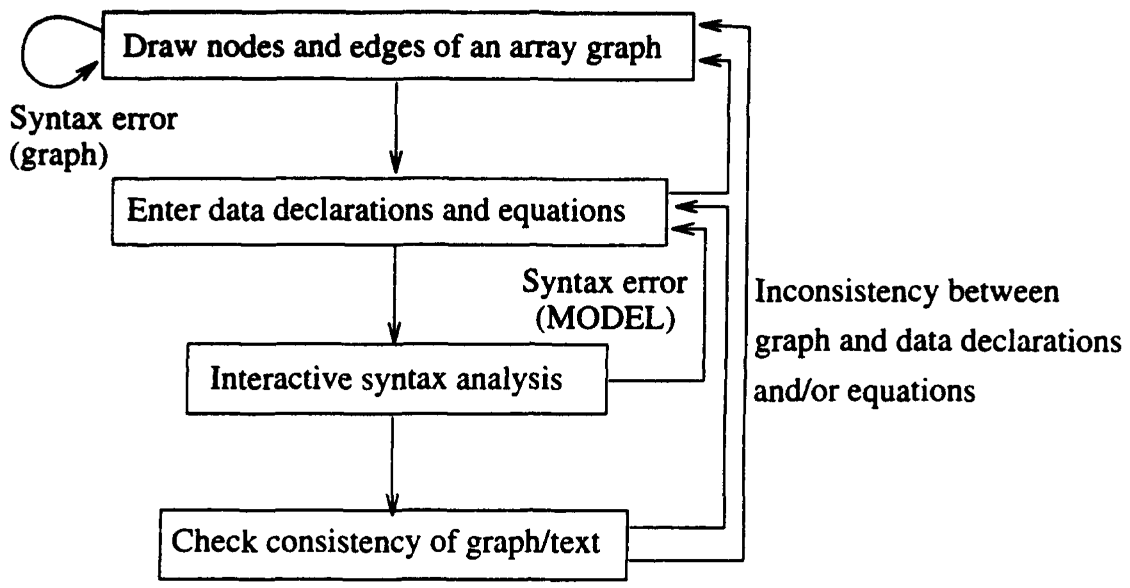

Draw nodes and edges of an array graph
Syntax error (graph)
Enter data declarations and equations
Syntax error (MODEL)
Interactive syntax analysis
Check consistency of graph/text
Inconsistency between graph and data declarations and/or equations

Figure 2.6: Composing, parsing, and checking an array graph.

data declarations and algebraic definitions of equations in textual forms via a form window. The graph editor is combined with a MODEL parser to perform interactive syntax analysis on the entered data declarations and equations. A user corrects syntax errors which are detected during the analysis. Consistency between the graph and the text are then checked. A user may correct the graph, the declaration, or the equations to correct any inconsistency detected during the checking. The interactive procedure of composing, parsing, and checking an array graph is illustrated in Figure 2.6. Chapter 3 discusses the more details of the visual programming.

2.5 Checking

A composed array graph with data declarations and equations can be directly translated to a MODEL specification in a textual form. Then the MODEL compiler generates an excutable code from the MODEL specification. During the compilation, the compiler checks if the specification satisfies the requirements of MODEL such that every variable must be defined by a unique equation. The MODEL compiler detects errors and warns a user if satisfying the requirements of a MODEL

Message Type	Explanation
Error	A problem in the specification is severe. No code is generated.
Failure	A problem internal to the checking system is found.
Limit	The specification requires an allocation beyond what is presently allowed.
Warning	The checking system makes an assumption about the specification. Verify the cause of the warning and determine whether the assumption is the intended meaning. Modify and recompile accordingly.

Table 2.1: Classes of Error Messages

specification is conditioned on the values of input data. The user must correct the graph according to the messages. Note that a user composes a MODEL specification by drawing an array graph. Therefore the error/warning messages must be presented in a graphical form.

The following stages are performed on the specification on which the syntax analysis has been already performed:

- Precedence[4] Analysis: The compiler examines the data dependencies[5] and the hierarchical dependencies[6] in the specification to complete an extended version of a symbol table called *dictionary* [Lu81]. The system finds incompleteness of naming variables and equations and produces error/warning messages.

- Dimension Propagation: Dimensionalities of variables are propagated throughout equations and variables with respect to data dependencies. This stage of checking detects missing subscripts of variables and inconsistencies in definitions and references of dimensionalities for variables. The subscript expressions (the labels of the connectors such as `(i)` and `(i-1)`) in Figure 2.1 are obtained during the dimension propagation stage.

- Range Propagation: The compiler identifies all subscripts of the same range

[4]Data and hierarchical dependencies.
[5]The data connectors of the array graph in Figure 2.1 visualize the data dependencies.
[6]The hierarchy connectors in Figure 2.1 denote the hierarchical dependencies.

(the maximum value of a subscript) in this stage. The information is essential to find an efficient execution sequence of the specification. Missing ranges of the subscripts and discrepancies among them are reported.

The error/warning messages are classified as shown in Table 2.1. The nodes and the edges of the visualized array graph, which are related to the messages, are graphically emphasized to facilitate users' understanding of the messages. Responding to the warning/error messages, a user fixes the faults in the specification. Chapter 4 discusses the details of the checking mechanism.

2.6 Testing

An equational specification is modified (possibly automatically) and recompiled before it is tested. The modified program execution stores the values of interim variables and facilitates tracing exclusive paths from input nodes to output nodes. A *complex conditional equation* is expanded each to a set of *simple conditional equations.* The conditions in the set of simple conditional equations must be exclusive. Equations are also added to define explicitly the conditions in the simple conditional equations. The modified equational specification is then compiled for the testing. An array graph is generated for the visualization. Executable code is generated and used for executing the tests.

The tester selects the conditions, one by one, from the array graph. Next, test input values are selected to satisfy the selected condition. The equational specification in then executed with the test input values. The nodes of the conditions, which have been previously selected and tested, are cumulatively shaded. The more conditions are satisfied during tests, the more condition nodes are shaded in the graph. This visualizes the progress in test coverage. The testing process ends when the test adequacy criterion is satisfied. This may mean that all conditions required by the adequacy criterion are shaded. The adequacy criterion may require one or multiple test input values that satisfy each condition. Note also that each test input value set may cause the shading of more than one condition node. In some test adequacy criteria, conjunction of conditions must be satisfied. A condition table is constructed to list the conjunctions.

The input data variables are obtained by backtracking, along the causality paths, from the selected condition to the respective input variables. The test input values are chosen from the domains of the input variables that satisfy the condition. Determining values of input variables that satisfy the condition may be complex. This is an assumed human operation which finds the values of the input variables satisfying the conditions. This is an "Oracle" assumption [Bei90]. If the conditions are not satisfiable, then the tester reports the developer to fix the problem. The input data are entered by pointing to the nodes of the associated input variables of the array graph. Respective files for input variable values are then created.

The compiled specification is executed with each selected test input values, shading condition nodes and producing the test results. The output values from the test execution can be viewed via pointing to their corresponding variable nodes. Note that a time-out is used to terminate the test if execution time is unreasonably long.

It is assumed that the human tester will be able to determine whether or not the output values from a test are correct. This is also an oracle assumption [RW85, Bei90]. If the results are incorrect, the tester reports the errors to the developer so that the equational specification can be fixed and then testing is restarted. Otherwise, the testing process continues until the test adequacy criterion is satisfied.

The more details of the testing method are described in Chapter 5.

2.7 Program Verification

Verification of the correctness of a MODEL specification is simple and intuitive, because it utilizes only algebraic manipulation of equations, namely equational reasoning. It is based on simple deduction rules by which equations can be deduced

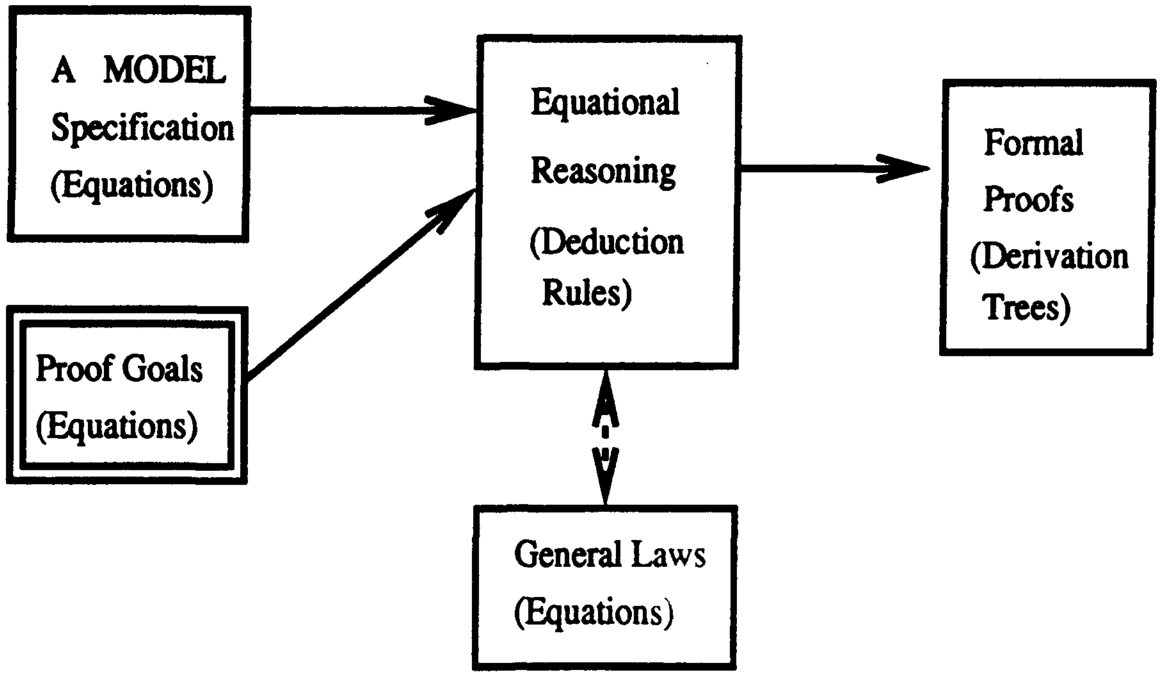

A MODEL Specification (Equations)
Proof Goals (Equations)
Equational Reasoning (Deduction Rules)
Formal Proofs (Derivation Trees)
General Laws (Equations)

Figure 2.7: An overview of a process of the program verification.

from equations and general algebraic laws. MODEL calculus (an equational reasoning system for MODEL) specifies the general algebraic laws and the deduction rules [Kim91].

Figure 2.7 illustrates the system for program verification based on equational reasoning: Given a MODEL specification which is a collection of equations, a user provides proof goals. The proof goals are represented as MODEL equations. The verification system (called MODEL calculus) is equipped with general algebraic laws (also represented as equations) and deduction rules. From the MODEL specification and the general algebraic laws (a set of equations) the verification system tries to derive each equation of the proof goals using the deduction rules. If the system can deduce an equation of the proof goals, the equation is said to be proved. The derivation tree, which records the derivation of the equation, becomes a formal proof of the equation. Otherwise, the equation is not proved. The derivation trees (the formal proofs of the equations in the proof goals) are output of the reasoning system.

Equations of a MODEL specification are regarded as axioms during the verification. Thus the followings are required:

- A user must assure that all the inputs to each equation are available.
- Every equation must compute the unique value of its LHS variable by evaluating its RHS expression.
- There must exist at least one causal chain [GR89] from input to output via the equations in a MODEL specification.
- The execution of a specification must terminate with a solution.

The verification process aims to prove the correctness of the specification as presented in the proof goals. The proof goals may consist, in the simplest case, of constraints on inputs and their expected outputs. They may consist of axioms (expressed as MODEL equations) about computational properties of the specification. Proof goals are decomposed into a collection of subgoals that can be proved through equational reasoning. Chapter 6 contains more details and an example.

2.8 Knowledge Acquisition

Many widely used programs in procedural languages contain a great deal of knowledge such as algorithms and methods which can be accumulated as rules of a rule-base expert system. The knowledge acquisition in the environment is based on language translation. Figure 2.8 illustrates an overview of the knowledge acquisition method. First, the procedural programs are translated to MODEL specifications. An equation of the MODEL specifications can be regarded as an assertion or a rule. Thus, the MODEL specifications are translated to rules in a rule-based language. This method offers an easy way in which knowledge bases of a rule-based expert system are enriched.

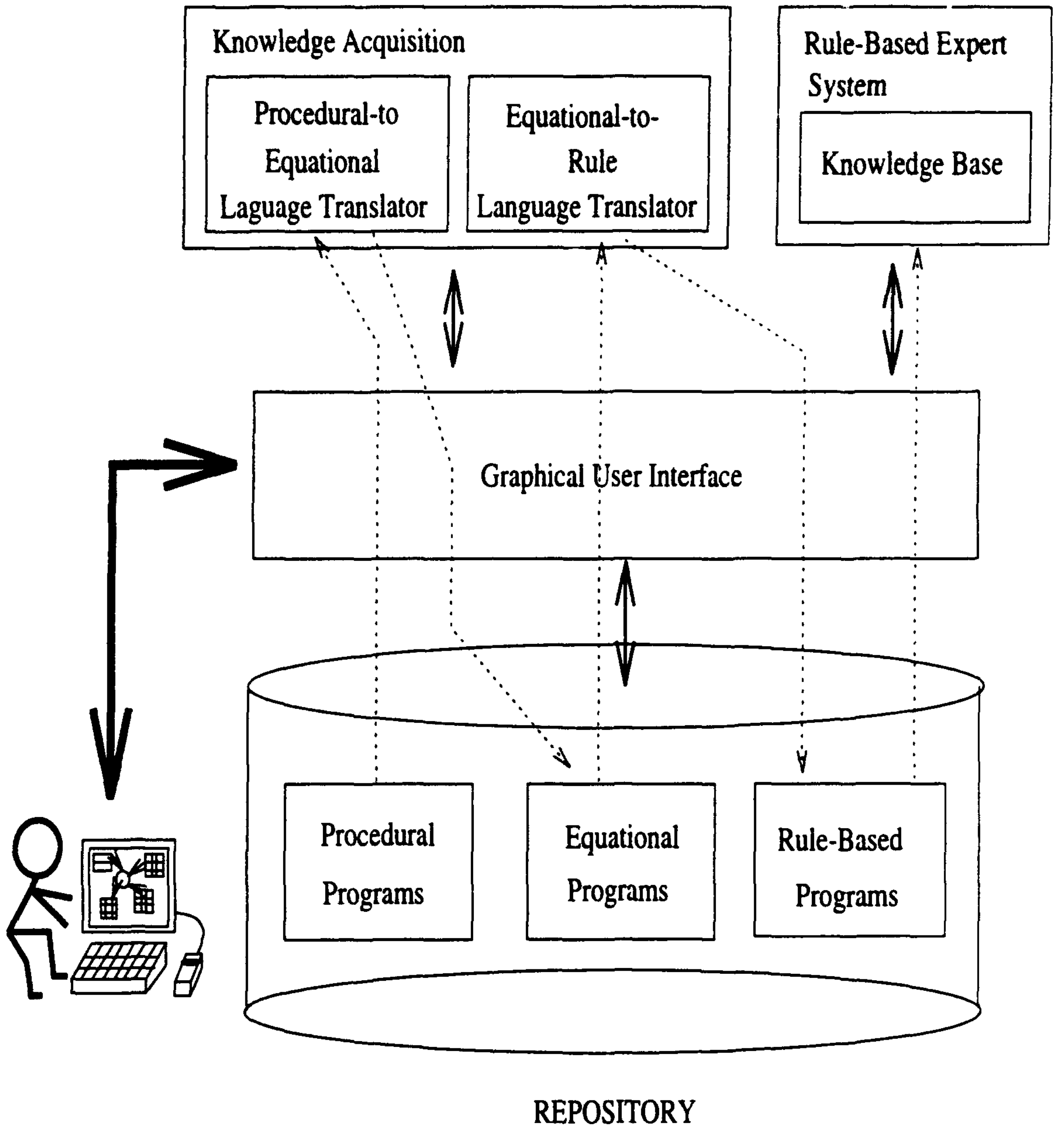


Figure 2.8: Rule extraction for a rule-based expert system from programs.

We must ensure that (1) the translation from the procedural programs to the MODEL programs does not change algorithms and methods in the procedural programs and (2) the translation from the MODEL programs to the rules maintains algorithms and methods in the MODEL programs. [PGLS90] has already shown that the procedural-to-equational translation is successfully implemented and works. We must show that the equational-to-rule translation can maintain algorithms and methods in this section. It is demonstrated by showing a fact that a rule of a rule-based language can simulate an equation of MODEL.

A rule of a rule-based expert system can be expressed as `precondition` ⇒ `action`. If a precondition of a rule is satisfied, an action is performed in rule-based programming. On the other hand, a MODEL equation is executed only if its existence condition is satisfied. Thus execution of an equation can be envisaged as inference of a rule; its existence condition can be regarded as a precondition of a rule: its equation body as an action of a rule. The rule is represented as `existence_condition` ⇒ `equation`. Thus a MODEL specification consisting of equations can be viewed as a set of rules. An input variable in a MODEL specification such as "`x`" with a value 3 can be interpreted as a fact (or relation) of a rule-based expert system such as "`(define-fact (x 3))`".[7] An array variable like "`y(i,j,k) = 47;`" can be expressed as a fact that maps multiple fields of its subscripts to its value such as "`(define-fact (y i j k 47))`". As for an equation, the existence condition of its input variables can be checked like the precondition of a rule is checked. The execution of the action of a rule is equivalent to the execution of the equation. The execution of a specification (a set of equations) can be envisaged as inference (or reasoning) of rules on given facts in a rule-based expert system. The order of firing represents causal chain among the rules [GR89]. Since a specification is assumed to have at least one order of firing that leads to a solution, there is at least one causal chain for a given set of facts.

It concludes that the execution of a MODEL equation is similar to the execution

[7]We follow the syntax of a rule-based language, CLIPS[GR89] in this document.

of a rule in the expert system. Therefore, an array graph for a MODEL specification can be translated into the sequence of pattern matching scheduled by the Rete algorithm [For82, GR89] without losing algorithms and methods of the MODEL specification. The more details are discussed in Chapter 7.

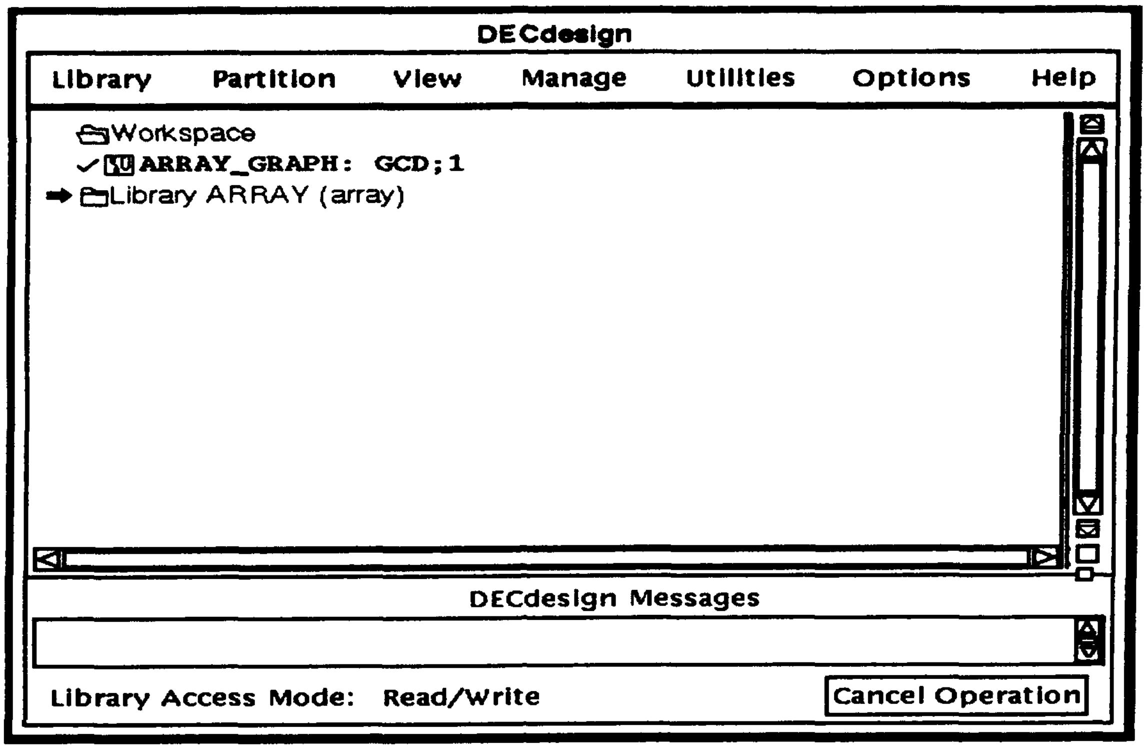
Chapter 3

Visual Programming

3.1 Introduction

The objective of this chapter is to describe an icon-based graph editor for visual programming of the environment. The icon-based graph editor is implemented using DECdesign which is a meta-environment based on graphics [DEC91]. It helps a user to analyze and design software systems according to sound rules of a design methodology. The implementation is exercised by providing the methodology of visual programming to the DECdesign core environment in terms of graphical objects and rules.

This chapter is organized as follows: The icon-based graph editor and its use in the composition via graph drawing are described in Section 3.2. As the first example, the Euclid algorithm of finding the Greatest Common Divisor (GCD) of two integers, which was presented in Chapter 2, is re-visited. Steps of defining data declarations and equations are explained in Section 3.3. Parsing data declarations and equations is discussed in Section 3.4. The composed graph and text must be consistent with respect to their meanings. The consistency checking of graph and text is discussed in Section 3.5. The graph can be used as graphical user interfaces for other functions such as static checking and testing which are discussed in Chapter 4 and Chapter



Workspace
✓ ARRAY_GRAPH: GCD;1
➡ Library ARRAY (array)

Library Access Mode: Read/Write

Figure 3.1: A DECdesign window.

3.2 Graph Drawing

3.2.1 Repository

A user opens a library where the visual programming methodology is encoded using Methodology Implementation Facility (MIF). A programming language of MIF is called Methodology Implementation Language (MIL) [DEC91]. The methodology is loaded on a workspace of DECdesign from the library. A user creates, saves, updates, and retrieves an array graph file to and from the repository of the environment. Figure 3.1 shows a user interface for managing the repository. It illustrates a status where the visual programming methodology called "ARRAY" is loaded and an array graph file ("ARRAY_GRAPH") titled "GCD" is created on the workspace ("Workspace").

The array graph file can be viewed, updated and saved into the repository using

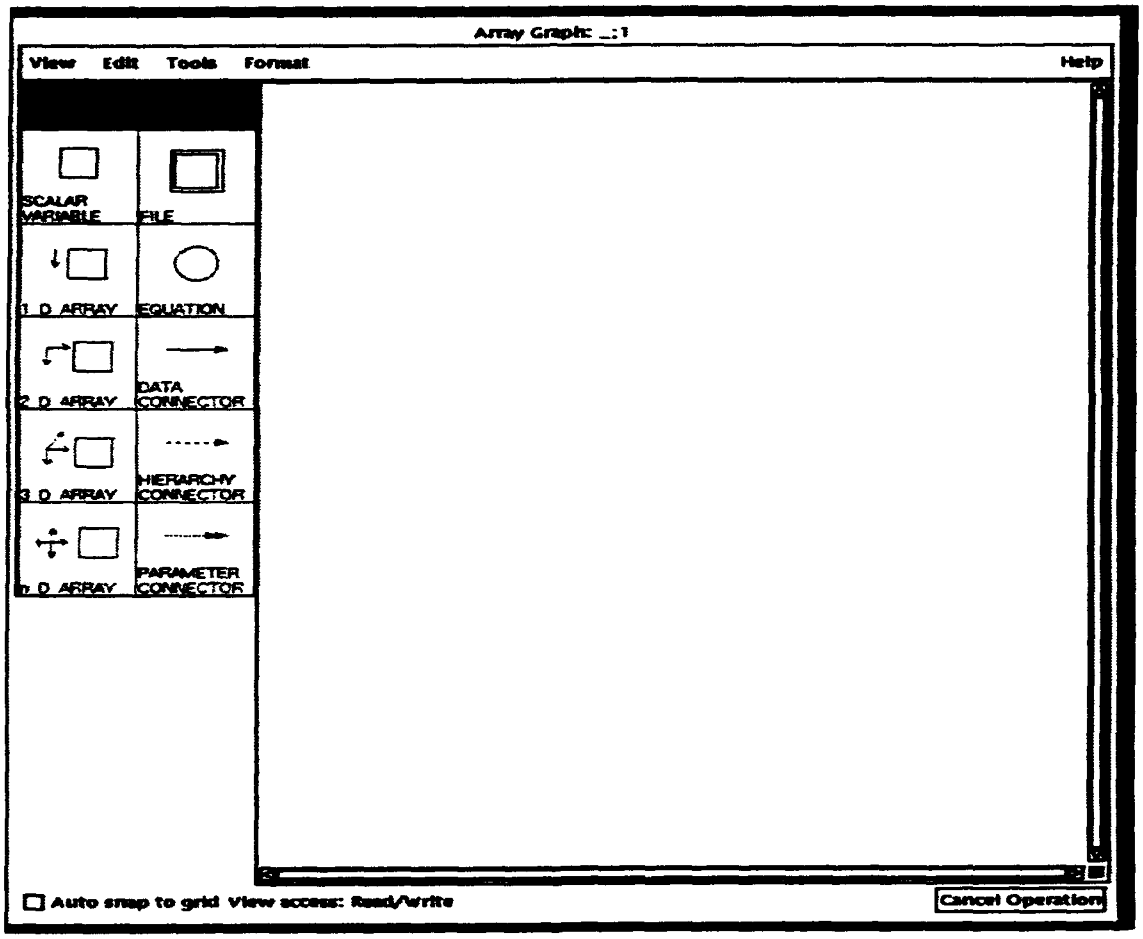


Figure 3.2: Icon-Based Graph Editor.

the icon-based graph editor illustrated in Figure 3.2. The editor has a control panel for selecting icons, a canvas for drawing graphs, pull-down menus, and scroll bars.

3.2.2 Icons

Symbols (nodes) and connectors (edges) of an array graph are denoted by icons. The icons are listed in a control panel of the icon-based graph editor as shown in Figure 3.2. The symbol icons denote files, records (physical units of communicating with external devices), groups (logical units of data), fields (physical memory cells containing values) or equations. The records, groups, and fields can be either a scalar or a multi-dimensional array. A scalar is denoted by a rectangle icon. Its name and node number are displayed in the rectangle icon. A multi-dimensional array is represented by an icon consisting of a rectangle and (an) arrow(s). A rectangle icon contains a name and a node identification number of a multi-dimensional array. The number of arrows denotes dimensionality of an array. For an array of more than 4 dimensions, the "n-D ARRAY" icon with 4 arrows is used. Arrows are annotated by their associated subscripts. For example, a three-dimensional array, `x(i,j,k)`, has three arrows which are annotated with subscripts, `i`, `j`, and `k`, respectively. As will be discussed in Section 3.2.3, a user only provides a name for a symbol. Its node number is automatically assigned by the icon-based graph editor. As the editor parses the node name, it finds the associated subscript and annotates arrows with their associated subscripts.

The icons without a shaded or a striped background like `x1, x2, y1(i)`, etc. in Figure 2.1 denote fields. Records and groups are shaded like the icons, `in_rec` and `out_rec`, in Figure 2.1. A user selects an icon (either a scalar or an array) from the control panel and specifies that it is either a record or a group. The editor analyzes the information and shades the icon. The details will be discussed in the next section. Control variables are striped like `END.y1(i)` (node number 9) in Figure 2.1. The editor recognizes the symbol as a control variable when it parses its name.

The inside of the icon is striped by the editor if its name has a prefix like `END` and `SIZE`.

Data connectors (arrows of a solid line) represent data dependencies between an equation symbol and fields. Hierarchy connectors (arrows of a broken line) denote hierarchical data dependencies among fields, groups, records, and files.

Entity-relation relationships among icons are as follows. An equation symbol can be connected to field symbols via a data connector and vice versa. Two different field, group, record, or file symbols can be connected to each other via a hierarchy connector and vice versa. The more detailed relationships are presented in Table 3.1: The "FILE" symbol can represent either an input or an output file. When it denotes an input file, it can be connected to either input record symbols or input group symbols via hierarchy connectors. Otherwise, there are hierarchy connectors from either output record symbols or output group symbols. The "SCALAR VARIABLE" symbol can be a field, a group, or a record. If the symbol is neither shaded nor striped, it denotes a field and can be connected to and from an equation symbol via a data connector. If the symbol is an input field, it can be connected to an input record or an input group via a hierarchy connector. If it is an output, a hierarchical connector goes to either an output record or an output group from the symbol. A shaded "SCALAR " symbol represents either a record or a group. It can connected to other fields, groups, or records via hierarchy connectors. Note that a record of records is not allowed in MODEL while a record of groups, a group of records, and a group of groups are possible. A control variable is treated as same as a field is, except that its associated icon is striped. The relational information about array symbols and an equation symbol is described in Table 3.1. All relationships are tested to find any syntax error in the graph by the editor whenever symbols and connectors are created.

Symbol	Color	Meaning	Direction	Other Symbols	Connector
FILE	none	input file	→	input rec, grp	h conn
		output file	←	output rec, grp	h conn
SCALAR VARIABLE	none	scalar variable	↔	equation	d conn
			←	input rec, grp	h conn
		(field)	→	output rec, grp	h conn
	shaded	scalar variable	→	input rec, grp, fld	h conn
		(record, group)	←	output rec, grp, fld	h conn
	striped	scalar variable	↔	equation	d conn
			←	input rec, grp	h conn
		(control var)	←	output rec, grp, fld	h conn
1-D ARRAY	none	1-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(field)	→	output rec, grp	h conn
	shaded	1-d array	→	input rec, grp, fld	h conn
		(record, group)	←	output rec, grp, fld	h conn
	striped	1-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(control var)	←	output rec, grp, fld	h conn
2-D ARRAY	none	2-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(field)	→	output rec, grp	h conn
	shaded	2-d array	→	input rec, grp, fld	h conn
		(record, group)	←	output rec, grp, fld	h conn
	striped	2-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(control var)	←	output rec, grp, fld	h conn
3-D ARRAY	none	3-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(field)	→	output rec, grp	h conn
	shaded	3-d array	→	input rec, grp, fld	h conn
		(record, group)	←	output rec, grp, fld	h conn
	striped	3-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(control var)	←	output rec, grp, fld	h conn
n-D ARRAY	none	n-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(field)	→	output rec, grp	h conn
	shaded	n-d array	→	input rec, grp, fld	h conn
		(record, group)	←	output rec, grp, fld	h conn
	striped	n-d	↔	equation	d conn
		array	←	input rec, grp	h conn
		(control var)	←	output rec, grp, fld	h conn
EQUATION	none	equation	↔	fld	d conn

Table 3.1: Icons and their connectivities.

3.2.3 Editing

There is no specified sequence of creating symbols and connectors for editing an array graph. We assume that a user starts to define a header such as an array graph name, a source file name, and a target file of an array graph. A name of an array graph, a source file name, and a target file name should be defined via a separate *form window.*

Then a file symbol can be created on a canvas of an array graph window. A user clicks a file symbol from the control panel and moves a cursor to the canvas to locate it on the right place. With one more click, a file symbol is created on the designated position. Since this node is created first, its node number is assigned as 1. The editor manages node numbers of symbols. Its name must be defined by a user via a form window. The name must be consistent with one defined as a source file name. Otherwise, a warning message is issued by the editor.

A user can create a connector, either a hierarchy connector (a solid arrow named "HIERARCHY CONNECTOR" in the control panel) or a data connector (a broken arrow named "DATA CONNECTOR"). The following explains steps of creating a connector:

- A user selects a source symbol of a connector. Then, a number of attach points appear on the boundary of the selected symbol.
- A user chooses one of the attach points.
- After the selection, all the attach points of the symbol except the selected one disappear.
- A user selects a target symbol of the connector. Again, a number of attach points appear on the selected symbol.
- A user selects one of the attach points. Then, a connector is drawn from the selected attach point of the source symbol to the attach point of the target symbol.

3.2.4 GCD Example

The Euclid algorithm, which computes the Greatest Common Divisor (GCD) of two input numbers, can be encoded as a module named GCD in Figure 2.1. The input file, in_file, contains a record, in_rec, which has two input fields, x1 and x2. The output of the module, i.e. the GCD of x1 and x2, is another field named z. Its value is contained in a record, out_rec, of an output file, out_file. Such hierarchies among fields, records and files are represented by hierarchy connectors as shown in Figure 2.1.

The Eq 1 equation determines values of the interim variable, y1(i). The equation needs values of x1, x2, y1(i), and y2(i). The data dependencies among the Eq 1 equation and the variables are expressed by data connectors as shown in Figure 2.1. The detailed definition of the equation is provided via a separate form window as will be discussed in Section 3.3. Similarly, the Eq 2 equation determines values of the interim variable, y2(i). Data dependencies among Eq 2 and its inputs and output are visualized. The variables, y1(i) and y2(i), contain partial results of the Euclid algorithm. The sizes of the variables are limited by a control variable, END.y1(i), computed by Eq 3. The one-dimensional array icon denoting the control variable is striped as shown in Figure 3.3. Finally, the GCD of the inputs, z, is calculated by Eq 4.

A user does not have to specify any sequence of execution or control. He just defines variables, equations, and hierarchies and data dependencies among variables and equations. A composed array graph denotes an equational specification in MODEL. An array graph must satisfy the following requirements: (1) Every variable must be uniquely defined and has a single value. (2) Every equation is fired as soon as all values of its inputs are available.

3.3 Data Declarations and Equations

A user creates symbols and connectors. Detailed data declarations and algebraic definitions of equations are typed in via separate form windows. For a data symbol, a form window is created. A user specifies a name of a symbol, its sort (field. group, or record), its data type, etc. For the equation symbol form window, name, comments, algebraic definition in MODEL, etc. are typed by a user.

3.4 Interactive Syntax Analysis

After drawing an array graph and entering required information for the graph and symbols, a syntax analysis is interactively performed. The purposes of the syntax analysis are:

- Detecting syntax errors with respect to the MODEL grammar.
- Completing a dictionary (an extension of a symbol table) which will be used for checking consistency of graph/text (Section 3.5) and the static checking (Chapter 4).
- Finding subscripts of arrays and annotating the corresponding array symbols with the subscripts.
- Recognizing records and groups to shade them.
- Recognizing control variables to stripe them.

A graph resulted from the analysis is shown in Figure 3.3. If there is a syntax error, the syntax analysis stops and an error message is issued for each symbol as shown in Appendix C.

3.5 Consistency Checking of Graph/Text

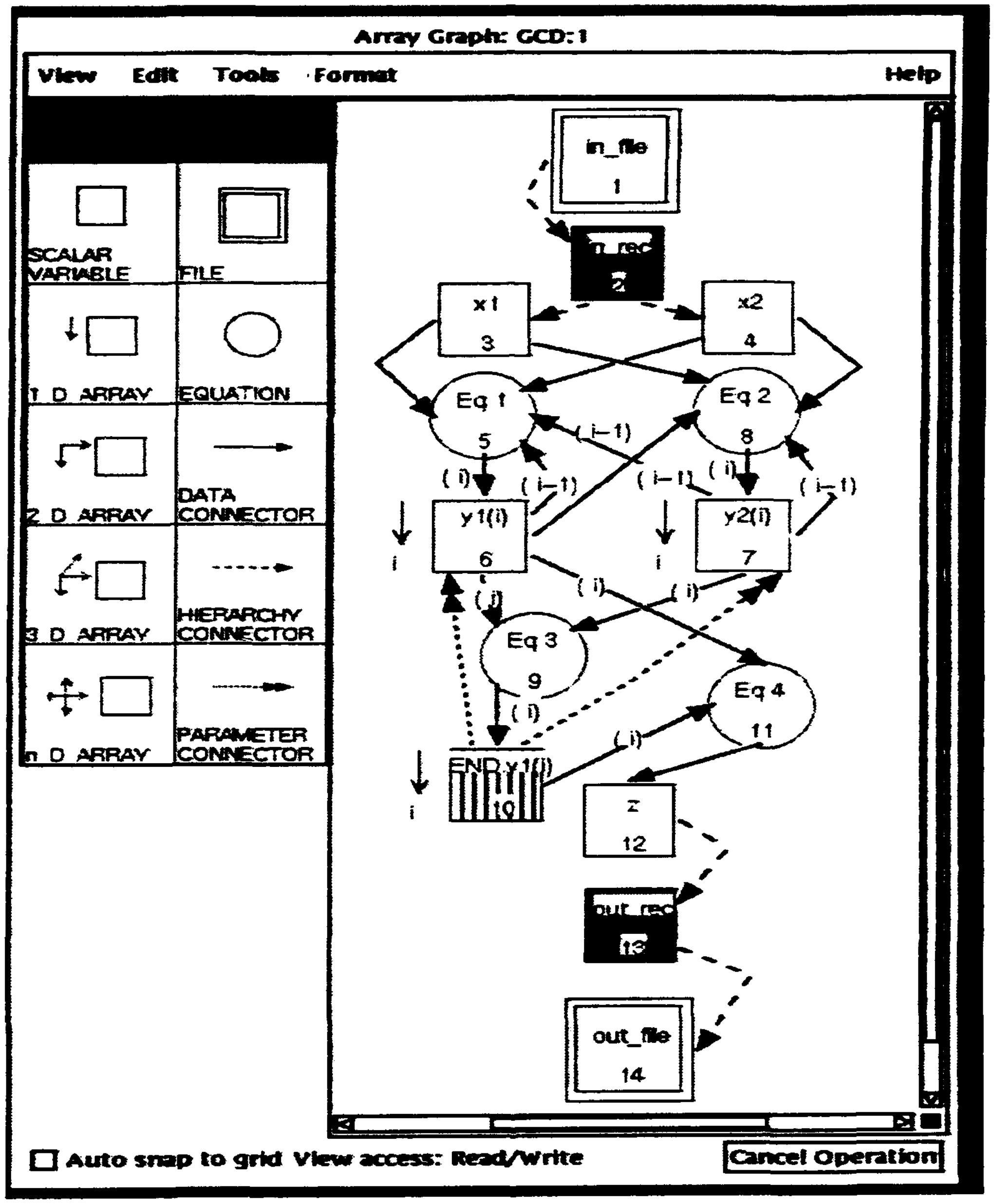


Figure 3.3: An array graph after the syntax analysis.

Id.	Message
1	Missing MODULE NAME
2	Missing SOURCE FILE NAME
3	Missing TARGET FILE NAME
4	Conflict in SOURCE FILE NAME
5	Conflict in TARGET FILE NAME
6	Conflict in INPUTS of EQUATION
7	Conflict in OUTPUT of EQUATION
8	Dimension Mismatch: Misuse of Scalar Icon
9	HIERARCHY_CONNECTOR cannot attach to EQUATION
10	HIERARCHY_CONNECTOR cannot connect SYMBOL to ITSELF
11	DATA_CONNECTOR cannot connect DATA SYMBOL to DATA SYMBOL
12	DATA_CONNECTOR cannot connect SYMBOL to ITSELF

Table 3.2: Error messages from the consistency checking mechanism of graph/text.

An array graph must satisfy the following requirements:

- A data symbol and its name must be consistent with each other in their dimensionality.
- An equation symbol can have multiple inputs (incoming data connectors) but a single output (an outgoing data connector).
- Data declarations of data symbols and their references in equation symbols must be consistent.
- A hierarchy connector cannot connect a data symbol and an equation symbol.
- A hierarchy connector cannot connect a data symbol to itself.
- A data connector can only connect a data symbol and an equation symbol.
- There is only one data connector between a data symbol and an equation symbol.
- A data connector is annotated with a subscript expression.

Those requirements are checked. If there is any inconsistency, warning/error messages are issued as specified in Table 3.2.

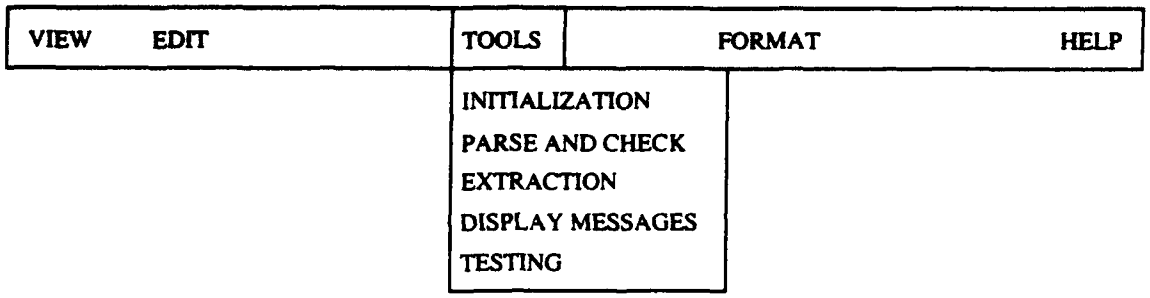


Figure 3.4: Tools menu.

3.6 Other Functions

A visualized array graph serves as graphical user interfaces for static checking and visual testing. A user can exercise the checking and the testing by clicking them from the "TOOLS" menu presented in Figure 3.4. The details of the checking and the testing will be discussed in Chapter 4 and Chapter 5, respectively.

3.7 Gauss Elimination Example

The Gauss Elimination algorithm, used as the second example in this chapter, is illustrated in Figure 3.5. The figure illustrates inputs and outputs as sets of simultaneous linear equations. The algorithm manipulates input matrix of coefficients and right hand side constants through successive matrices until it obtains a matrix with the lower left hand triangle of zeroes. A solution for the equations can then be readily obtained.

Figures 3.5 portrays graphics for the Gauss Elimination example. The figure shows how the input matrix, `m_i(i,j)`, is translated into a series of matrices that form a cube, `a(i,j,k)`, with two associated arrays: `q(i,k)` denotes nonzero elements and `p(k)` denotes positions of pivoting elements. Finally, the output, `m_o(i,j)`, is presented. Its associated text is given in Figure 3.6.

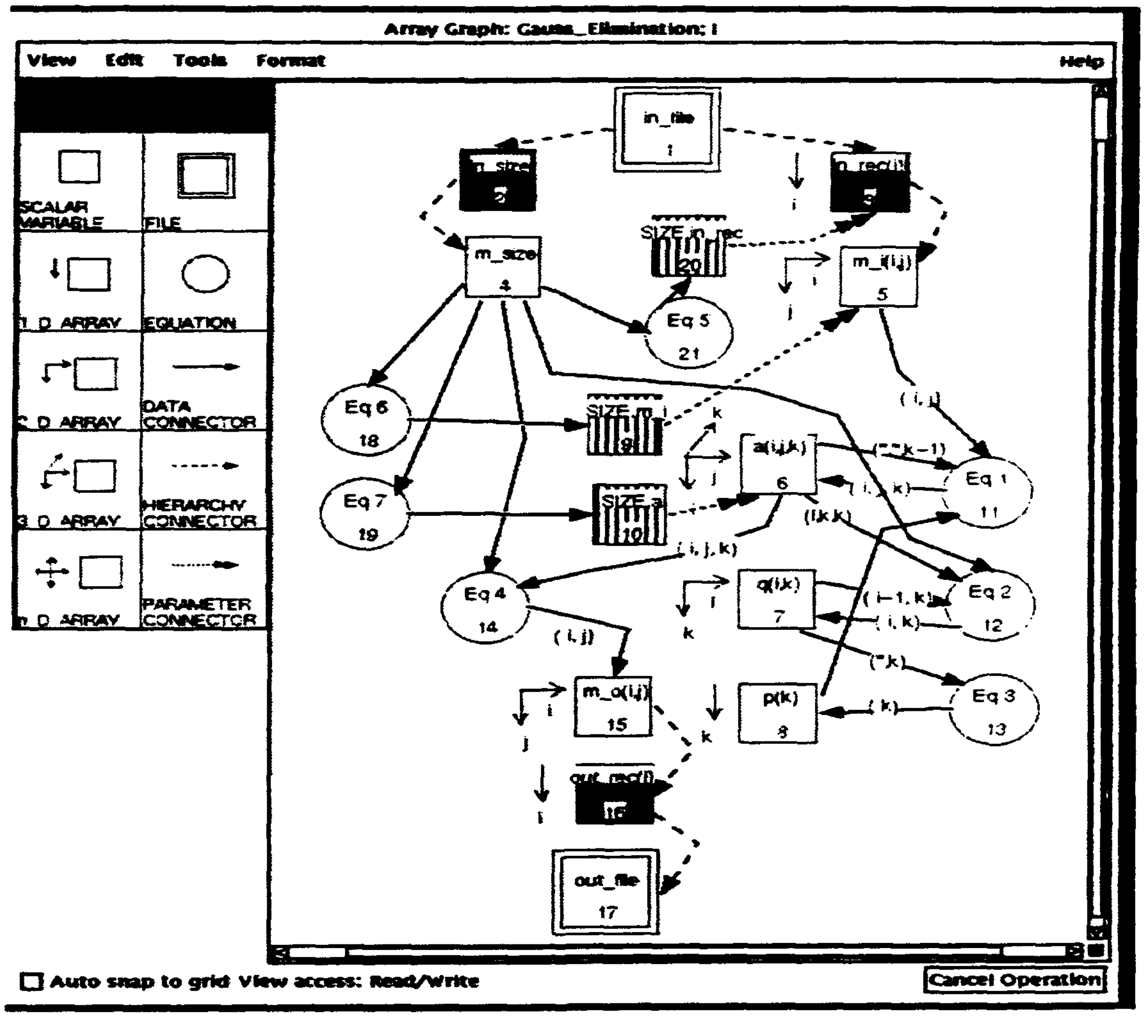


Figure 3.5: Gauss Elimination: array graph.

```
/* Header */
MODULE: Gauss_Elimination;
SOURCE: in_file;
TARGET: out_file;
/* Data Declaration */
/* Source */
1  in_file IS FILE,
  2  in_size IS RECORD,
    3  m_size IS FIELD (pic '9'),  /* size of matrix */
  2  in_rec(1:100) IS RECORD,
     3  m_i(1:100) IS FIELD (pic '---9v.9');  /* input linear system */
/* Target */
1  out_file IS FILE,
  2  out_rec(*) IS RECORD,
    3  m_o(*) IS FIELD (pic '---9v.9');  /* output, triangular matrix */
/* Interim variables */
1 aaa(*) IS GROUP,
  2  aa(*) IS GROUP,
    3  a(*) IS FIELD (pic '---9v.9');
/* Subscripts */
(i,j,k) ARE SUBSCRIPTS;
/* Equations */
/* Eq 1 */
a(i,j,k) = IF k=1 THEN m_i(i,j) /* load input matrix */
           ELSE IF p(k-1)=0 THEN 0 /* no pivoting, no solution */
           /* switch pivoting row p(k-1) with (k-1)th row */
           ELSE IF i=p(k-1) THEN a(k-1,j,k-1)
           /* switch (k-1)th row with pivoting raw p(k-1) */
           ELSE IF i=(k-1) THEN a(p(k-1),j,k-1)
           ELSE IF i<(k-1) THEN a(i,j,k-1) /* no need to pivot */
           /* zeroing the lower left triangle */
           ELSE a(i,j,k-1) - a(p(k-1),j,k-1)*a(i,k-1,k-1)/a(p(k-1),k-1,k-1);
/* Eq 2 */
q(i,k) = IF i=1 THEN IF a(i,k,k)=0 | k>1 THEN 0 ELSE 1
         ELSE IF i<k THEN 0 ELSE IF a(i,k,k)=0 THEN q(i-1,k) ELSE 1;
/* Eq 3 */
p(k) = IF k=1 & i=1 & q(i,k)=1 THEN i
       ELSE IF i>1 & k<=i THEN IF q(i-1,k)=0 & q(i,k)=1 THEN i;
/* Eq 4 */
m_o(i,j) =  IF k=m_size THEN a(i,j,k);
/* Eq 5 */ SIZE.in_rec = m_size;
/* Eq 6 */ SIZE.m_i = m_size+1;
/* Eq 7 */ SIZE.a = m_size;
```

Figure 3.6: Gauss Elimination: text.

Chapter 4

Compilation

4.1 Introduction

This chapter describes the compilation part of the environment where static checking is performed for equational specifications in MODEL. A user composes a MODEL specification by drawing an array graph in an array graph window using an icon-based graph editor to state equations and declarations. This was discussed in Chapter 3. Syntax and semantics of an equational specification can be checked. We will use the static checking mechanism of syntax and semantics of MODEL in the MODEL compiler [Lu81]. The environment utilizes the compiler for checking of a composed equational specification. The MODEL compiler has been developed for a MODEL specification in textual form. An array graph composed using an icon-based graph editor in graphical form must be transformed into a MODEL specification in order to utilize the compiler. Though an array graph and a MODEL specification can be functionally equivalent, there is a gap in their appearances, i.e., graphs and texts. The gap becomes wider due to a fact that messages from the compiler are in textual form and based on a text-based MODEL specification. For example, a message refers to an equation of a specification while a user wants to locate a node in a graph which represents the equation. To reduce the gap, the environment does (1) conversion of an array graph to a MODEL specification before the checking (2) visualization of messages in the array graph in order to make a user interpret them easier.

The compiler checks a user in composing a MODEL specification that complies with semantic requirements of MODEL. The requirements are (1) existence of all variables of equations in a MODEL specification (2) existence of at least one causality chain from input to output[GR89] via equations of a MODEL specification. Thus a unique solution for equations must be always computable for a given input set. A corollary requirement is that its execution must terminate. The compiler performs static checking that detects errors or warns a user when satisfying these requirements is conditioned on values of input data. The compiler also adds missing statements or parts of statements. Responding to the messages, a user fixes faults in a specification.

The compiler exercises the following:

- **Ambiguous** definitions and **incomplete** definitions of variables and equations are detected as their array graph and dictionary [Lu81, MOD89] are completed:[1] If a same name is used for more than one data structures, it causes ambiguity. It must be resolved by using qualifying (or prefixing) names for variables [Lu81, SLPP84]. The incomplete definition of an LHS variable of an equation means that it does not have an RHS expression that defines its value.

- The **existence requirement** for defining variables is checked: It is examined by checking definitions of variables and their references. There may be discrepancies between declarations and references about dimensionalities, data types of variables and ranges of subscript expressions. They are checked by propagating attributes such as dimensions and ranges (sizes of dimensions of array variables) via edges of an array graph.

- **Causality** that computes a solution set for a given input values is checked:

[1] A dictionary is an internal representation of a specification. Header, detailed declarations of variables, equations and their precedence relationships are stored in a dictionary.

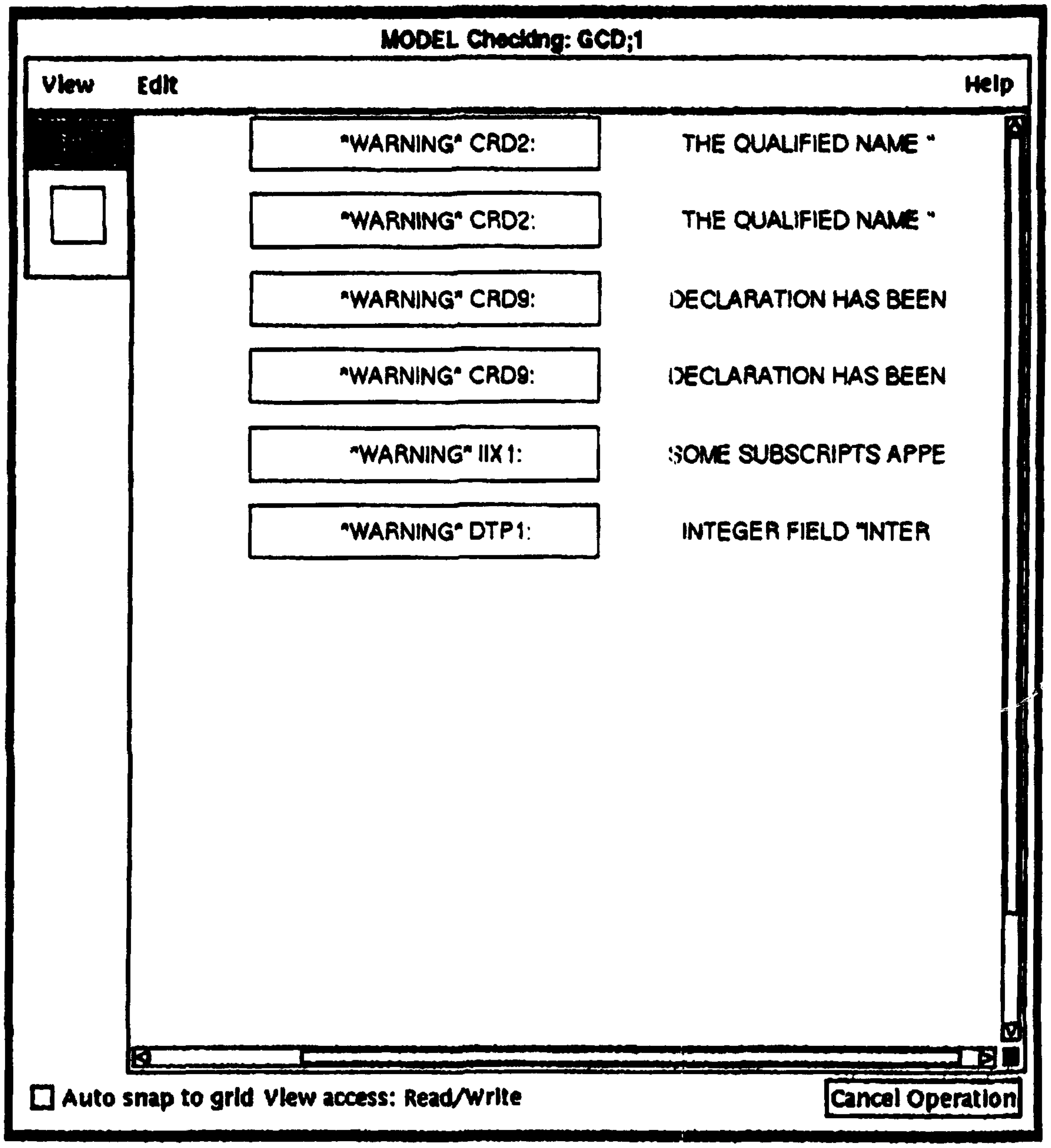
A cyclic definition of variables (called *circular logic* [SLPP84]) may cause an infinite computation. It should not be a part of any causality chain. The checking mechanism detects such a cycle in an array graph and tests if it results in a cyclic definition.

- The condition of **terminating** execution is checked for a specification: Even though there is no circular logic in an array graph, it may not terminate if MODEL control variables specifying its termination condition such as END do not have finite values. The presence of such control variables and their computability are checked. If they are not found on a causality chain or their values cannot be determined as finite, a warning message is generated.

An example of viewing messages from the compiler is illustrated in Section 4.2. Section 4.3 describes a method of discovering the ambiguous and the incomplete definitions. Checking the existence requirement is discussed in Section 4.4. The method of detecting and removing the cyclic definitions of variables is presented in Section 4.5. Checking the termination condition is discussed in Section 4.6.

4.2 Example

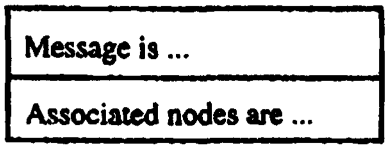
Checking the GCD example shown in Figure 3.3 illustrates the environment. Checking a MODEL specification is initiated by user's calling the MODEL compiler. The compiler generates warning/error messages. Since a user is dealing with a visualized array graph, a method of displaying messages must be consistent with the method of composing an array graph. A user selects "Display messages" from the pull-down menu of "Tools" in an array graph window as shown in Figure 3.4 in order to display messages from the checking. They are displayed as message icons (boxes) on a separate message window as illustrated in Figure 4.1. A class of a message such as "ERROR", "FAILURE", "LIMIT", and "WARNING" and a mnemonic of a message such as "CRD2", "CRD9", "IIX1", and "DTP1" are printed within a message



MODEL Checking: GCD;1

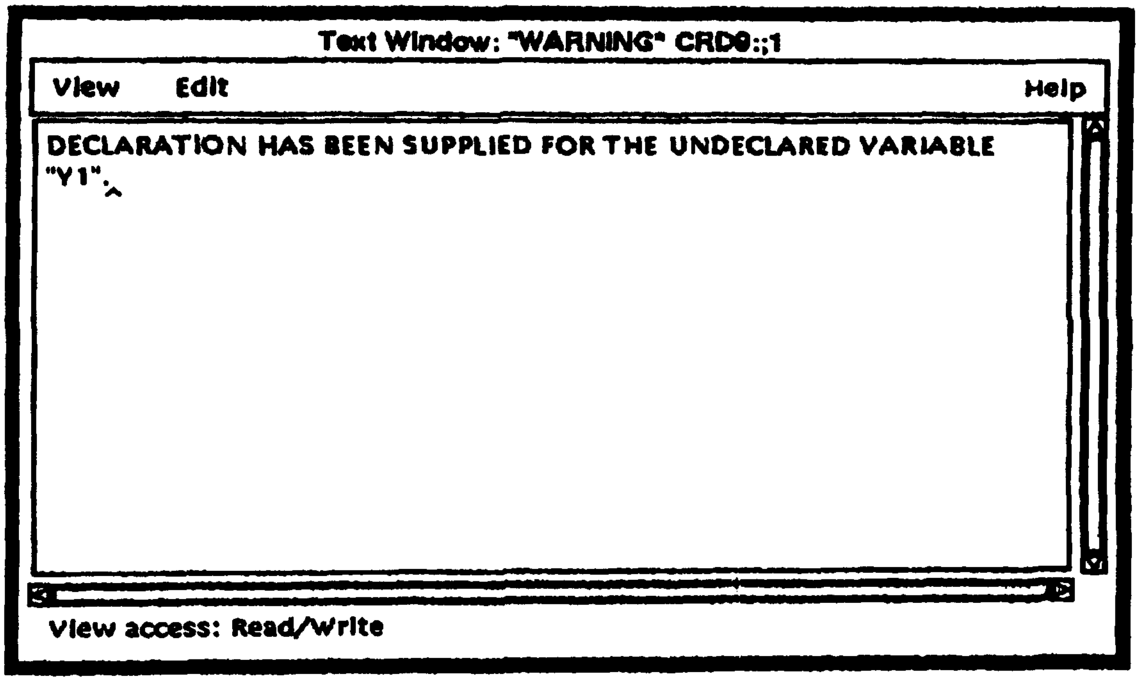
WARNING CRD2: THE QUALIFIED NAME "
WARNING CRD2: THE QUALIFIED NAME "
WARNING CRD9: DECLARATION HAS BEEN
WARNING CRD9: DECLARATION HAS BEEN
WARNING IIX1: SOME SUBSCRIPTS APPE
WARNING DTP1: INTEGER FIELD "INTER

Figure 4.1: Message window.



Message is ...
Associated nodes are ...

Figure 4.2: Pop-up menu for Message.



DECLARATION HAS BEEN SUPPLIED FOR THE UNDECLARED VARIABLE "Y1".

Figure 4.3: Content of message labeled "*WARNING* CRD9:".

icon.[2] A fraction of a message is printed next to a message icon. A user can either browse contents of messages or locate nodes and edges of an array graph which are associated with messages. Content of a message can be examined by selecting the entry of "Message is..." from the pop-up menu shown in Figure 4.2. Figure 4.3 shows a message window of the third message, "*WARNING*:CRD9:", of the message window shown in Figure 4.1. If a user wants to locate nodes and edges of an array graph, which are associated with messages, he must select "Associated nodes are..." from the pop-up menu in Figure 4.2. The associated nodes and edges can be searched and visually highlighted as shown in Figure 4.4. At the same time, a small dialog box as shown in Figure 4.5 is popped up in order to inform a user that the highlighted nodes and edges are associated with the message.

[2]For the details of checking message mnemonics and their meanings, see [MOD89].

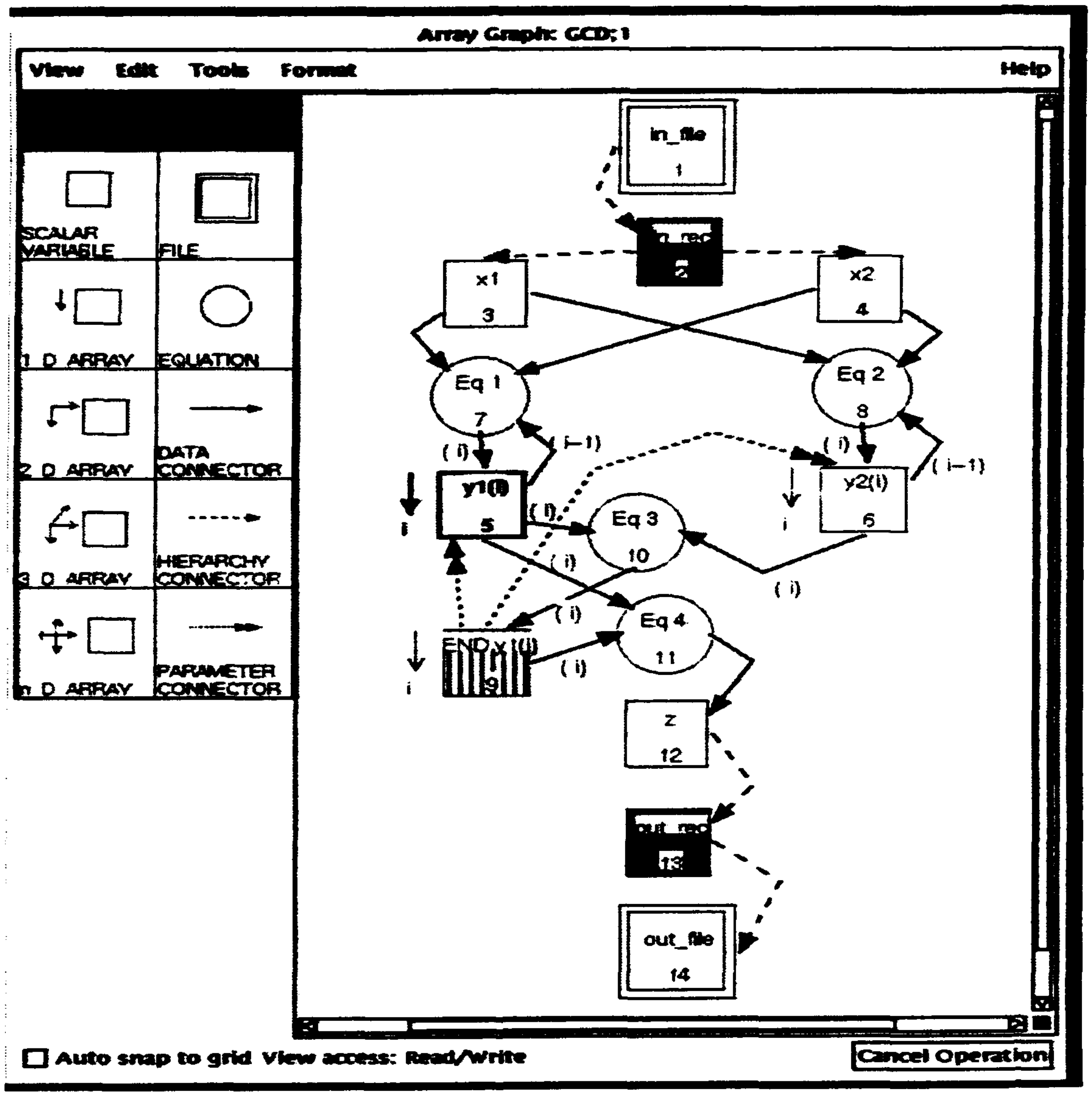
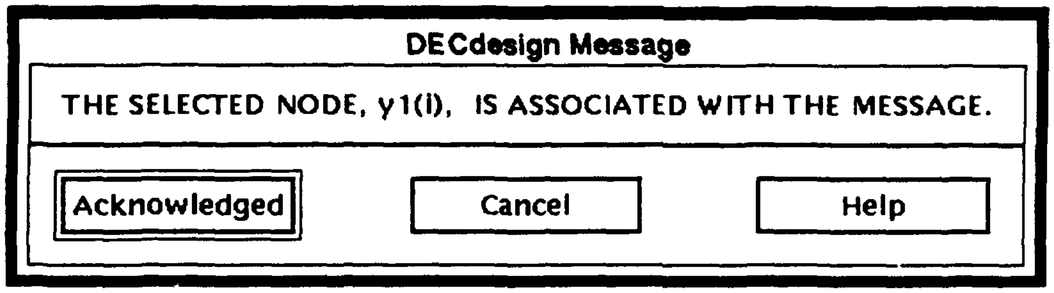
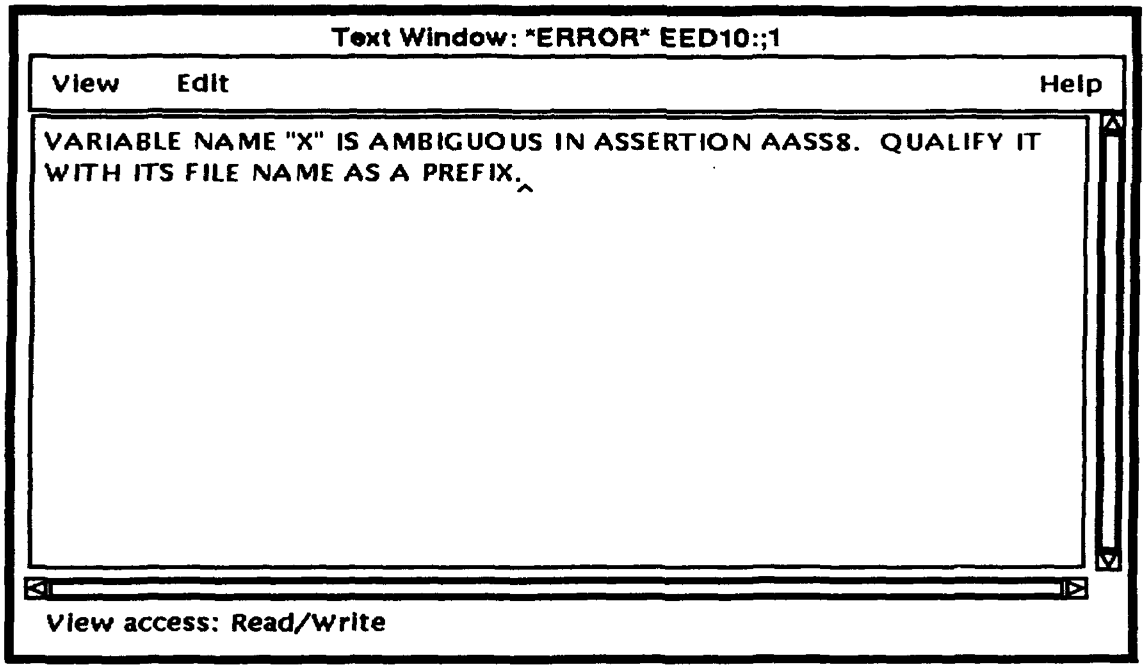


Figure 4.4: Node y1(i) is highlighted in the array graph window.



THE SELECTED NODE, y1(i), IS ASSOCIATED WITH THE MESSAGE.

Figure 4.5: A user is informed that y1(i) is highlighted.



VARIABLE NAME "X" IS AMBIGUOUS IN ASSERTION AASS8. QUALIFY IT WITH ITS FILE NAME AS A PREFIX.

Figure 4.6: Ambiguous definition is detected.

4.3 Dictionary Construction

Every variable and its definition are listed in a dictionary of a MODEL specification. As a dictionary is constructed by the compiler, ambiguous definitions and incomplete definitions of variables are detected. A warning message is issued for ambiguous or incomplete definitions.

4.3.1 Ambiguous Definitions

When several data structures have the same name, it is ambiguous to reference the data structures from equations. The ambiguity can be partially removed as a dictionary is constructed. It is done by applying the following rules [Lu81, SLPP84]:

- An LHS may reference only interim or output variables.
- An RHS may reference also input variables.

In many cases, however, a user is required to remove ambiguity. He may rename ambiguous variables by qualifying (or prefixing) them [SLPP84]. For example, we may have the following declaration statements:

```
1 a IS GROUP,                          1 b IS GROUP,
  2 x(i) IS FIELD NUM(4);                2 x(i) IS FIELD NUM(4);
```

The following equation has ambiguity because of the field **x**:

```
y(i) = IF x(i)>10 THEN x(i) - 10
                  ELSE x(i);
```

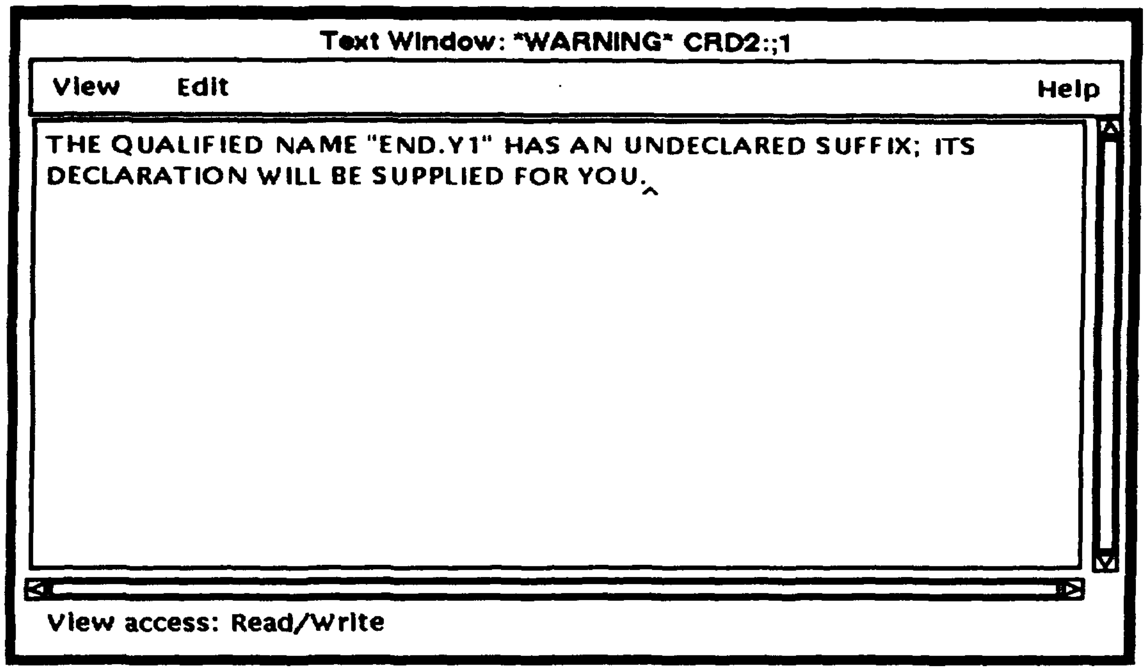
The compiler would detect this ambiguity and issue a message shown in Figure 4.6. Such ambiguity can be resolved by using qualified (or prefixed) names as follows:

```
y(i) = IF a.x(i)>10 THEN b.x(i) - 10
                    ELSE b.x(i);
```

4.3.2 Incomplete Definitions

If equations or interim data declarations are omitted, the compiler attempts to provide an appropriate equation or a data declaration. The process is based on the following rules [Lu81, SLPP84]:

- If an output data node is not explicitly defined, a new equation may be composed using its implicit input nodes.



```
THE QUALIFIED NAME "END.Y1" HAS AN UNDECLARED SUFFIX; ITS
DECLARATION WILL BE SUPPLIED FOR YOU.
```

Figure 4.7: Incomplete definition is detected.

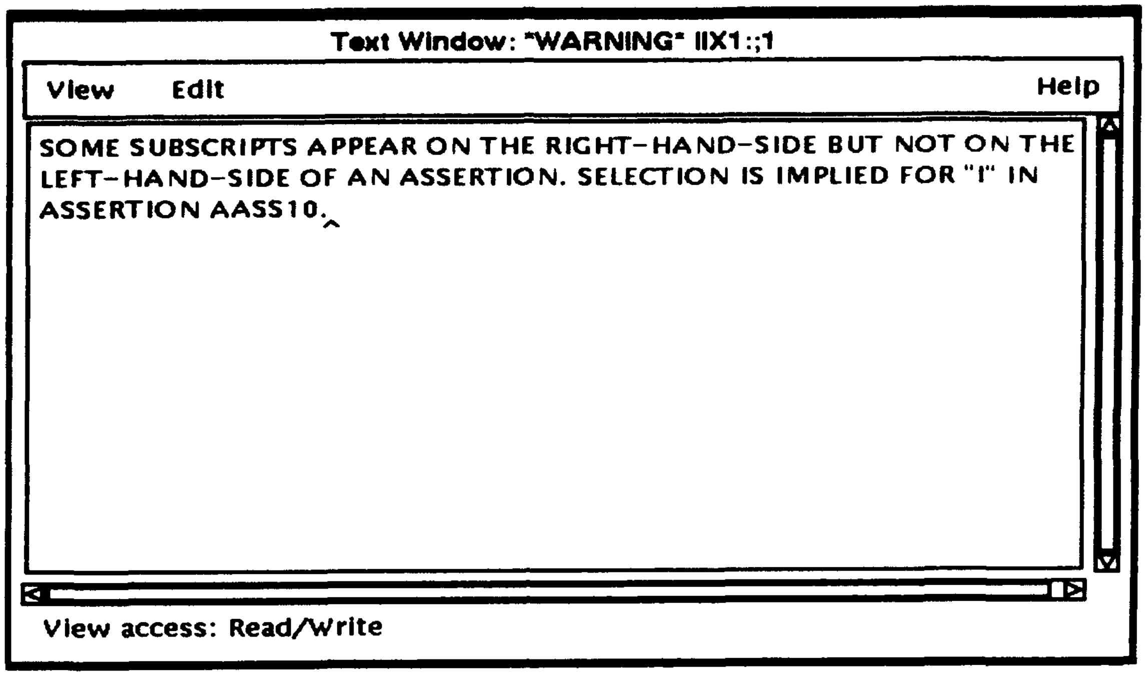
- An omitted data declaration of a node (an interim variable) and/or its parent node can be formulated using its implicit inputs.

If the implicit source of the omitting equations and the declarations are not found in an array graph and/or a dictionary, the system requests a user to provide equations and/or data declarations.

The message shown in Figure 4.7 informs a user (1) the system detects an undeclared variable, `y1`, while it checks a qualified variable, `END.y1`, (2) the system will provide a default declaration for `y1`.

4.4 Existence Requirement

The existence requirement of variables is the most important property that a MODEL specification must employ. To facilitate a user to comply with the requirement, the compiler provides utilities that examine the consistent definitions/references of dimensions and their ranges.



```
SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR "I" IN
ASSERTION AASS10.
```

Figure 4.8: Incomplete definition of a subscript is detected.

For each equation of a specification, an RHS expression, $f(i_1, ..., i_n, j_1, ..., j_k, var_1, ..., var_k)$, for instance, and its input variables, $var_1, ..., var_k$, have consistent definitions of their dimensions. An LHS variable, $x(i_1, ..., i_n)$, has consistent definitions of dimensions with respect to an equation. The dimension propagation algorithm checks consistency in definitions of dimensions and their references through out equations of a specification. Then ranges (sizes) of their dimensions are propagated along dependency edges of their array graph. Their consistency is checked by the range propagation algorithms. As a result, the existence condition, namely, $\exists j_1, ..., j_m, var_1, ..., var_k$ and $\exists i_1, ..., i_n$ is checked.

The warning message presented in Figure 4.8 is generated during checking of the existence condition for the following equation:

```
z = IF END.y1(i) THEN y1(i);
```

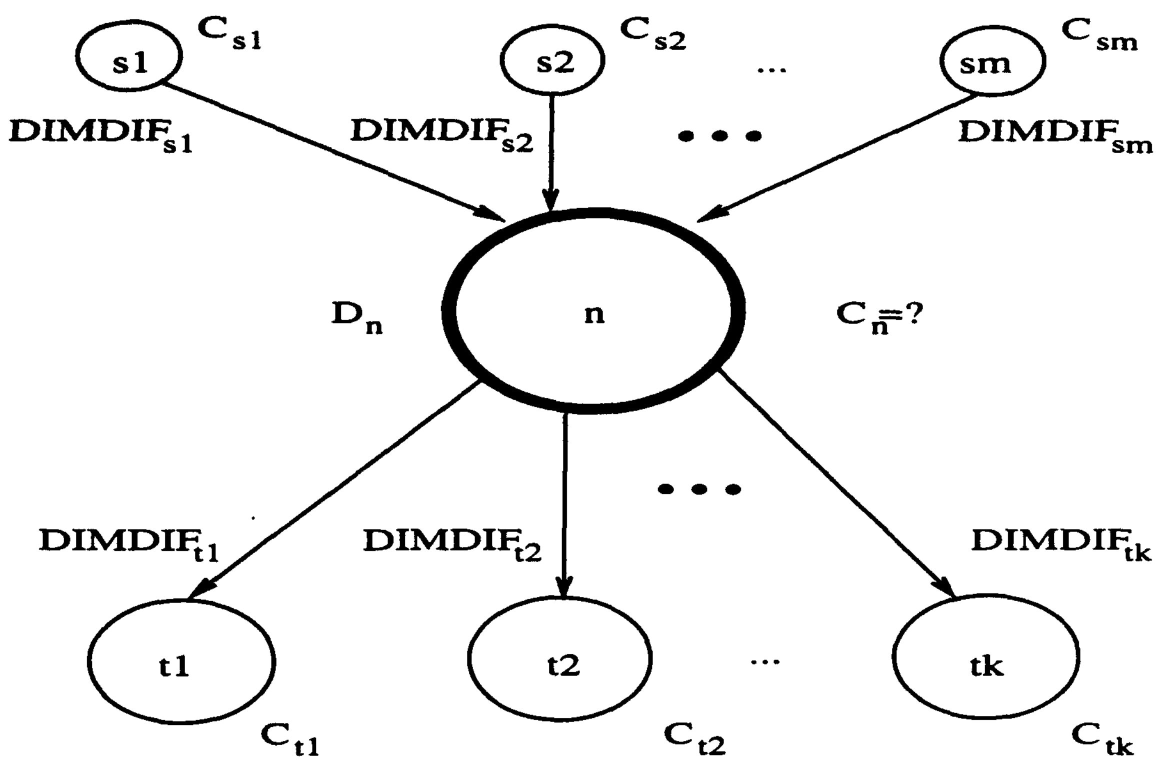


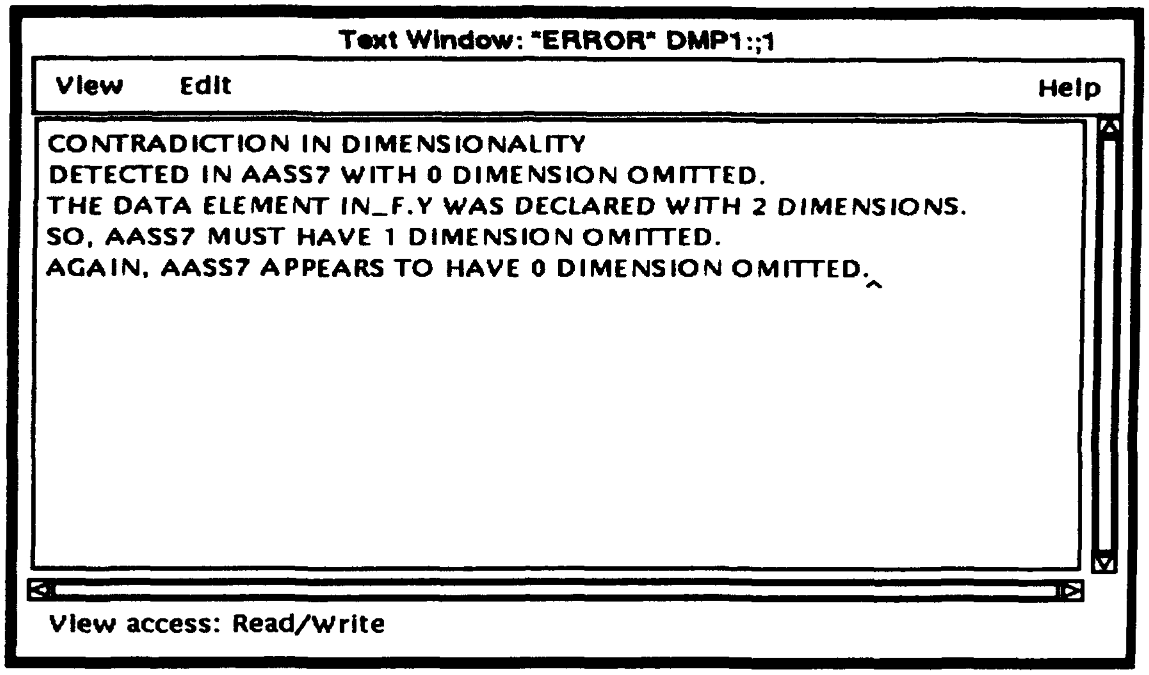
Figure 4.9: Dimension Propagation.

4.4.1 Dimension Propagation

A user does not have to specify a detailed dimensionality of a variable in MODEL. It is necessary to check for the compiler if dimensionalities of arrays referenced in equations are consistent with that of those arrays specified in their respective data declaration. The compiler completes data declarations and equations whose dimensionalities are not explicitly specified. This is done by dimension propagation.

The compiler propagates attributes of a node of an array graph to another via an edge that connects the nodes. The attributes of an edge stored in a dictionary include the following [Lu81, Ge89]:

- a source node of an edge.
- its target node.



```
CONTRADICTION IN DIMENSIONALITY
DETECTED IN AASS7 WITH 0 DIMENSION OMITTED.
THE DATA ELEMENT IN_F.Y WAS DECLARED WITH 2 DIMENSIONS.
SO, AASS7 MUST HAVE 1 DIMENSION OMITTED.
AGAIN, AASS7 APPEARS TO HAVE 0 DIMENSION OMITTED.^
```

Figure 4.10: An error is detected by dimension propagation.

- its type, i.e., hierarchical precedence, data dependency or parameter precedence.
- difference of the source and the target nodes in their dimensionality (DIMDIF).
- its subscript expression list.
- range set for each dimension.

If two nodes are linked by an edge, attributes of the nodes must be matched according to attributes of the edge.

An algorithm for dimension propagation is described in [Lu81]. Dimensionality differences, $DIMDIF$, are set up for all edges of an array graph. For input file nodes, their dimensionalities are 0. An intermediate node, n, (either a variable or an equation) has an initially declared number of denoting its dimension, D_n. Suppose m source nodes, $s_1, ..., s_m$, are connected to a node, n, via respective coming edges of dimension differences, $DIMDIF_{s_1}, ..., DIMDIF_{s_m}$. There may be

k target nodes, $t_1, ..., t_k$, connected by k outgoing edges of dimension differences, $DIMDIF_{t_1}, ..., DIMDIF_{t_k}$, as shown in Figure 4.9. Current dimensionality of a node, x, is denoted by C_x: source nodes, $s_1, ..., s_m$, have dimensionalities, $C_{s_1}, ..., C_{s_m}$ and target nodes, $t_1, ..., t_k$, have dimensionalities, $C_{t_1}, ..., C_{t_k}$. Then, dimensionality of the node n is defined as follows:

$$C_n = max_{1 \leq i \leq m, 1 \leq j \leq k}(D_n, C_{s_i} + DIMDIF_{s_i}, C_{t_j} - DIMDIF_{t_j})$$

The dimension propagation algorithm computes C_n for all nodes of the graph. If every node of the graph has a finite dimension, the algorithm converges [Lu81]. An infinite propagation cycle of an array graph can be detected by the algorithm. Then, nodes and edges on the cycle are revealed so that a user can fix it. If dimensionalities of all nodes and edges are correctly defined, an output file node of the graph must have 0 dimensionality.

Contradiction in dimensionalities of variables is detected by the demension propagation algorithm. Figure 4.10 displays an error message called "DMP1". A user declares an input variable, y, as a two-dimensional array and references it as a one-dimensional array in an assertion[3] named "AASS7".

Missing subscripts of equations can be filled up during the dimension propagation. A node subscript list is formulated for each variable node. Based on these lists, missing subscripts of equation nodes and missing subscript expressions of edges are filled up. The detailed procedure is described in [Lu81].

4.4.2 Range Propagation

After the dimension propagation, ranges (sizes) of dimensions are examined for all nodes in an array graph. The basic strategy is to find user specified ranges of nodes and propagate them to the rest of the nodes in the graph along with edges connecting them. The propagation aims:

[3]Denoted by an equation node in an array graph. A user may want to execute the "Associates nodes are..." operation from the pop-up menu in order to locate its associate equation node in the graph.

- to derive a range for a node subscript not having an explicit range.
- to determine range sets each of which contains two node subscripts of the same range.
- to check consistency in definitions and references of the ranges.

4.4.2.1 Definitions

A node subscript is defined for a node of an array graph as follows:

- $< x, i >$: a node subscript for an i-th (i is a positive integer) dimension of a node of an array variable, x.
- $< Eq_n, I >$: a node subscript for I (a subscript variable) with an equation node, Eq_n.

A range (or size) of a node subscript, $< n, d >$, is defined as $R(< n, d >)$.

4.4.2.2 User Specified Ranges

A range is specified explicitly or implicitly for each node. It may be explicitly defined by:

- a data declaration statement
- a subscript declaration statement
- values of control variables (`SIZE` or `END`)
- the system default: the end-of-file or end-of-record marker (`ENDFILE`) of an input sequential file

4.4.2.3 Condition of the Propagation

When two node subscripts of different nodes are related through a certain dependency relation and one of them does not have an explicit range specification, range of the other node subscript is propagated through the edge denoting the dependency relation.

If a subscript expression, $i - k$, where i is a subscript and k is a positive integer, is used in an equation, a mapping exists between values of elements indexed by i and $i - k$. It is assumed that a node indexed by i and an equation node indexed by $i - k$ are in the same range set.

4.4.2.4 Priority of the Propagation

There may be many alternatives to the range propagation. It is performed based on the following rules:

- All the node subscripts with the same global subscript[4] are considered as a single group, i.e., a set of variables and statements in a single loop in a procedural language program. Thus a range of a global subscript is propagated with the top priority.

- A data array node and its associated control variable such as `END` and `SIZE` must have the same range. A range of a control variable is required to be explicitly specified. A range is propagated from a control variable to its data array node with the second priority.

- A range of an output node is propagated to its associated equation node with the second priority.

- From an equation node to its associated input data node, a range can be propagated. It has the third priority.

[4]defined by either a subscript declaration statement or a control variable, **FOR_EACH** [Lu81, MOD89].

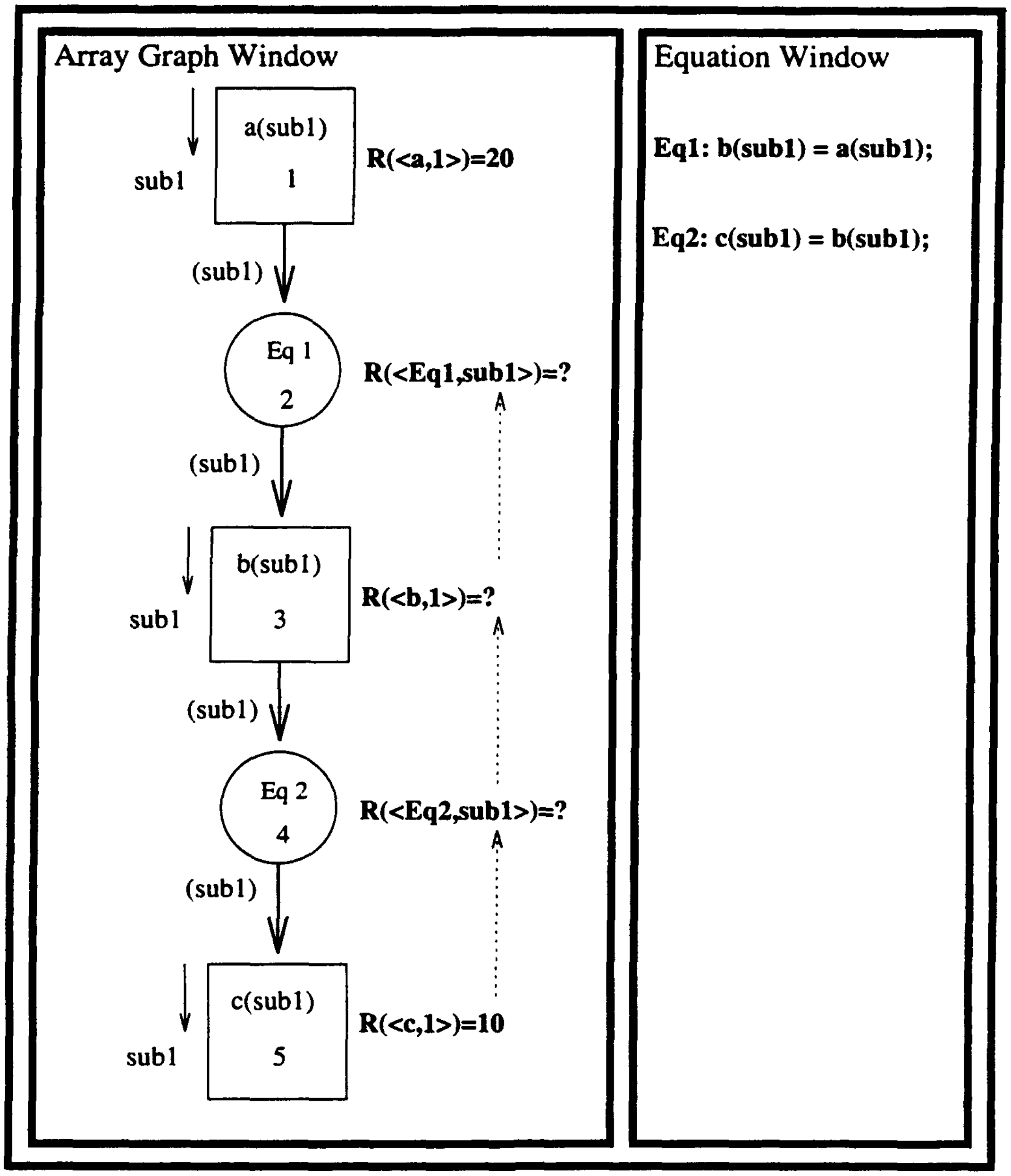


Figure 4.11: Example of Range Propagation.

- The lowest priority is given to the range propagation from an input data node to its equation node.

Consider a simple example illustrated in Figure 4.11. Two simple equations, Eq 1 and Eq 2, of transferring values from an array to another are presented. Ranges of the arrays, a(sub1) and c(sub1), are given as 20 and 10, respectively. Note that the subscript, sub1, is not defined as a global subscript. Since the condition of the range propagation is satisfied, we can propagate these ranges in order to determine the ranges of the node subscripts for Eq 1, Eq 2 and b(sub1). We have the following alternatives: (1) propagate the range of the local subscript sub1 of a(sub1) to the equation node, Eq 1, to determine the value of R(<Eq 1,sub1>) or (2) propagate the range of the subscript sub1 of the output node, c(sub1), backward to the equation node, Eq 2, to get the value of R(<Eq 2,sub1>). According to the rules of the range propagation, the first alternative, (1), has the fourth priority and the second alternative, (2), has the second priority. Thus the value of R(<Eq 2,sub1>) is defined as 10. Next, we have the following two alternatives: (1) and (3) propagate the value of R(<Eq 2,sub1>) to its input data node, b(sub1). The alternative, (3), has higher priority. Therefore the value of R(<b,1>) becomes 10. Finally, the following two alternatives remain: (1) and (4) propagate the range of the subscript, sub1, for the output node, b(sub1), to its equation node, R(<Eq 1,sub1>). The second alternative, (4), has the second priority. It follows that R(<Eq 1,sub1>) is equal to 10.

4.4.2.5 Range Functions and Real Arguments

A node subscript represents an iteration over its range by a loop control statement in a procedural program [Lu81]. An equation and a data node in an array graph may have multiple node subscripts and they represent a multi-level nested loop. In such a situation, a range of a node subscript can be a function of the other subscripts. For example, consider the following MODEL specification:

```
a IS FIELD;
b IS FIELD;


Eq 1: b(i,j,k) = a(i,j,k);
Eq 2: SIZE.a(i,j) = f(i,j);
```

The range of the third dimension, k, of the array variables, a(i,j,k) and b(i,j,k), depends on the ranges of the first and the second dimensions, i and j, as Eq 2 defines. The specification is translated into the following code of a procedural language program [Lu81]:

```
DO <a,1>;
   DO <a,2>;
      DO <a,3> = 1 TO SIZE.a(<a,1>,<a,2>);
         b(<a,1>,<a,2>,<a,3>) = a(<a,1>,<a,2>,<a,3>);
      END;
   END;
END;
```

An n-dimensional range array, SIZE.$x(i_1, ..., i_n)$, is regarded as a range function. The range function accepts integer arguments, $i_1, ..., i_n$, and computes the range of the n+1-th or higher dimension of the variable, x. Arguments of a range function are called real arguments, if they really contribute to determining a value of a function. An algorithm of finding real arguments of range functions is described in [Lu81].

It is required that loops of an array are nested according to a sequence of their array dimensions. That is, loops of a variable, $x(i_1, ..., i_n)$ must be nested in the following way:

```
DO <x,1>;
   DO <x,2>;
      ...
```

```
            DO <x,n>;
                ...
            END;
        ...
    END;
END;
```

It follows that a range function for a dimension, i_k, $1 \leq k < n$, does not affected by its lower dimensions for all i_m, $k < m \leq n$.

4.4.2.6 Range Propagation Algorithm

There are three basic algorithms for the range propagation. The first algorithm locates user specified ranges of node subscripts. As discussed before, ranges are specified by declaration statements (either data or subscript), control variables such as END and SIZE or the system default (end-of-file or end-of-record). Secondly, explicit range specifications are propagated. It requires node subscripts to be partitioned into their corresponding range sets. Finally, a real argument list is formulated for node subscripts in the same range set and is propagated. See [Lu81] for the details of the algorithms.

4.5 Causality Chain

A causality chain in an array graph is a path from its input nodes to its output nodes via its equations. A solution of equations is computed along such a causality chain. Therefore, a circular definition of variables, namely circular logic (or dependency), that causes an infinite computation, should not be on a causality chain.

A maximally strongly connected component (MSCC) of an array graph could result in circular logic. However, not all MSCC's form circular logic. The MODEL compiler identifies and decomposes an MSCC by deleting edges that represent data

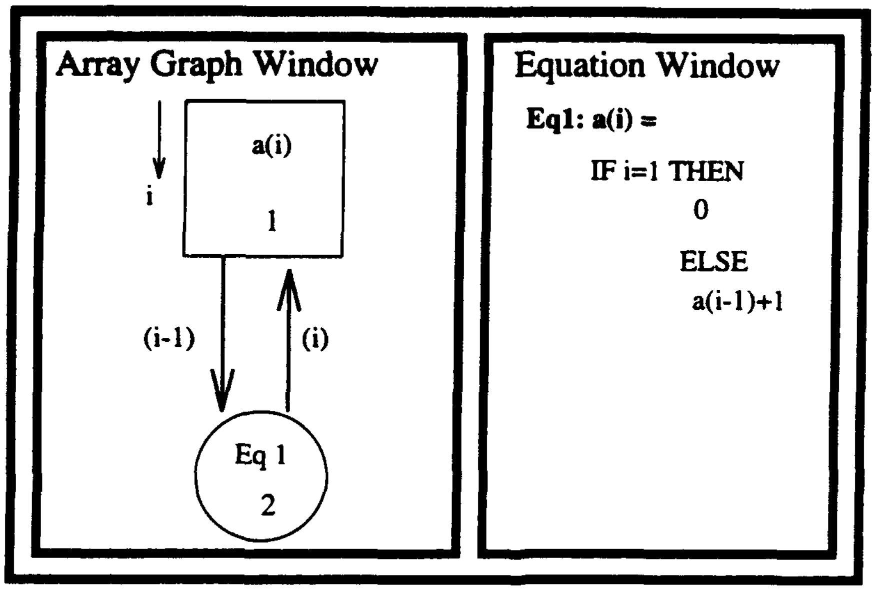


Figure 4.12: A maximally strongly connected component (MSCC) of an array graph.

dependencies assured by iteration statements [Lu81, SLPP84]. Such edges of iteration can be determined by examining subscript expressions of edges. If its subscript expression is in the form of $sub - k$, where $k > 0$ and sub denotes a subscript common to all MSCC nodes, an edge is determined to be representing iteration.

For example, an MSCC can be found in Figure 4.12: a cycle formed by the edge labeled by `i` and the edge labeled by `i-1`. In this particular example, $sub = i$ and $k = 1$. Therefore, the edge of `i-1` is classified as one representing iteration. It follows that the MSCC does not have circular logic. Such an iteration solution method is recursively applied until all MSCC's in an array graph are examined.

A cycle that cannot be decomposed by the iteration solution method is reported as a possibly infinite loop. A user has to examine and remove such a cycle by decomposing it. If it is not possible, he may use a set of simultaneous equations that perform the same function of the cycle. In general, it is very complicated to remove an infinite loop from a program by a static checking. We only deal with a specification that always has at least one causality chain of its equations.

4.6 Termination

Suppose we have an acyclic array graph without any cycle. Then a causality chain can be formulated. However, it does not mean that a solution of equations is obtained. To check the termination condition, equations defining control variables such as END and SIZE are examined for each causality chain. It aims to check *finiteness* of subscripts, namely, $1 \leq i_1 \leq \mathtt{SIZE}_{i_1}, ..., 1 \leq i_n \leq \mathtt{SIZE}_{i_n}$ and $1 \leq j_1 \leq \mathtt{SIZE}_{j_1}, ..., 1 \leq j_m \leq \mathtt{SIZE}_{j_m}$ of the existence condition. Values of SIZE variables (= ranges of subscripts), if computable, may be obtained during the range propagation. There may exist some constraints of defining ranges that cannot be computed during the range propagation. Such constraints are discovered and checked during the termination checking.

Note that an END variable can define a minimum range of 1 because it must have at least one boolean value. However, a SIZE variable can have a minimum range of 0. Value of an END variable can be infinite while a SIZE variable has a finite value. $\mathtt{END}.x(i_1, ..., i_n)$ may depend on values of the array, $x(i_1, ..., i_n)$. But $\mathtt{SIZE}.x(i_1, ..., i_n)$ must be computed before any element of $x(i_1, ..., i_n)$ is used.

The termination checking discovers an equation of defining sizes of array variables such as
"$\mathtt{END}.x(i_1, ..., i_n) = ...$" and "$\mathtt{SIZE}.x(i_1, ..., i_n) = ...$" for each causality chain of a specification. If ther[illegible] equation of such control variables, a warning message is issued so that a user [illegible]nes the termination condition of a specification. Though such an equation is defined on a causality chain, it may not have an explicit finite value denoting the termination. In such a case, a warning message is also issued.

Chapter 5

Testing

5.1 Introduction

The objective of this chapter is to describe a **software testing methodology** for the **Equational Visual Programming Environment** and discuss its theoretical background. The objective of software testing is to demonstrate that a program achieves its requirements.

The testing methodology presented in this chapter is called *Equational Visual Testing.* The testing methodology for procedural programs is called *Procedural Testing.* The approach to the Equational Visual Testing borrows from research into systematic testing of procedural language programs [DLS78, Nta84, RW85, Wey86, How87, DMMP87, Ham88, Bei90]. In both methods, the program (equational or procedural) is executed with a series of input values and the respective results are evaluated. Both the Equational Visual Testing and the Procedural Testing focus on analysis of graphs representing respective programs.

Exhaustive testing (using all possible combination of input values in the respective domains of the input variables) is impractical in both methods. Therefore, a compromise is made. Lesser testing requirements are imposed. However, the testing *needs to test systematically* sequences of program events. Such systematic testing is said to meet an *adequacy criterion.* Adequacy criteria vary in their order of strictness and have been shown experimentally to represent a similar order of program reliability. The stricter the criteria, the fewer errors go undetected.

Both testing methods are based on representing a program as a graph. They require selection of input values that force traversal of selected paths in the graphs. As will be shown, the selection of input values is based on satisfying conditions in the programs.

Thus the Equational Visual Testing and the Procedural Testing are somewhat similar. However, the semantics of the graphs differ considerably. The following differences are selected as significant to the labor required for testing:

(1) The Equational Visual Testing tests the equational specification without regarding to how it is executed, e.g., in a sequential or in a parallel computer. The Procedural Testing is based on the execution of the program in a sequential computer.

(2) The equational specification uses the language of MODEL and the **array graph** (See Chapter 2 for details.) A procedural program uses a sequential programming language and a **control flow graph** [Nta84, Wey86, Bei90]. Both methods require test executions that traverse selected classes of paths in the graph. Some of these paths, e.g., a path from a variable definition to its use, consists of a single edge in the array graph, while they may consist of multiple edges in the control flow graph. This may impact the difficulty of finding input values for testing.

(3) A single node in the array graph may represent an array variable or an equation defining multiple elements of an array variable. As noted, an array variable node denotes evaluation of all elements in the array. The evaluation of the elements may be sequential, parallel, or any other order that takes into account precedences due to dependencies. In contrast, cycles in the control flow graph represent sequential iteration.

(4) There may be also maximally strongly connected components (MSCC) in the array graph. As discussed in Chapter 4, the equational language compiler deletes edges resulting in a cycle-free array graph. As will be discussed, these deletion will allow transforming the array graph into an acyclic graph. In the control flow graph, all loops for defining or updating array elements are represented by an MSCC. Generally more MSCC's make it more difficult to analyze and select test input values.

(5) Equational languages have the single assignment rule stipulating that variables are defined only once and not modified. All variables are defined in every test execution. There are no undefined variable as in procedural programs. Thus all values of elements of arrays are available for analysis of execution behavior. In procedural languages, only the latest modification of a variable is preserved. The Equational Visual Testing process include a recompilation which preserves values of all input, output, and interim variables on secondary storage.

These differences and their significance will be further discussed in the sequel.

The plan for this chapter is as follows. Section 5.2 provides a background of the test methodology for procedural programs. It is based on the work of [Nta84, Wey86]. It describes use of control flow graphs and adequacy criteria to control the testing process.

Section 5.3 gives an overview of the Equational Visual Testing. This is based on the equational language and the array graph described in Chapter 2. It describes an approach to establish adequacy criteria similar to the ones used in the Procedural Testing. The remainder of the chapter discusses the steps of the Equational Visual Testing process described in Section 5.3.

Section 5.4 describes the pre-test compilation needed for the equational language. Its purposes are (1) to isolate conditions that control traversal of paths and (2) to retain array variable values needed for analysis of errors in the equational specifications.

Adequacy Criterion	Mean No. Test Cases	Bugs Found (%)
random testing	100	79.5
branch	34	85.5
all-uses	84	90.0

Table 5.1: Ntafos' experiment.

Section 5.5 presents a discussion of the adequacy criteria for the Equational Visual Testing and corresponding testing processes for satisfying these criteria.

The GCD example of Chapter 2 is used in Section 5.6 to illustrate the testing steps. The section discusses analyzing the array graph, selecting test input values, and evaluating the results of test execution.

5.2 Background: Procedural Testing

Testing a program for all the combination of all values in its input domains would assure that the program has no errors. Such exhaustive testing is impractical. As an alternative, a finite subset of values is selected from the input domains to satisfy requirements of an *adequacy criterion.* The program is executed with the selected test data. The test results are evaluated.

An empirical study by Ntafos has compared different test adequacy criteria applied for testing procedural programs [Nta84, Bei90]. As shown in Table 5.1, the *all-uses* criterion requires more test cases but finds more bugs than the *branch* criterion. The all-uses criterion requires fewer test cases than the random testing but finds more errors. These adequacy criteria are explained in Section 5.2.3. Another empirical study at IBM showed that the all-uses criterion has the best payoff with respect to the number of required test cases versus the number of detected errors [Bei90].

Theoretical considerations on the basis for adequacy criteria for the Procedural Testing of procedural language programs have been reported by Weyuker [RW85, Wey86]. It has been claimed that the use of adequacy criteria is a practical testing approach [Wey90]. The adequacy criteria are expressed in terms of paths in a control flow graph which is described in Section 5.2.1. The process of the Procedural Testing is presented in Section 5.2.2. Section 5.2.3 reviews the adequacy criteria theory for the Procedural Testing. It serves as the background for developing adequacy criteria for the Equational Visual Testing.

5.2.1 The Control Flow Graph

The control flow graph is also called *definition use graph* [RW85]. A node of the graph represents a *block* of procedural program statements. The block is a segment of a program with no branching within it. The first statement of the block is the entrance to the block. The last statement of the block precedes branches from the block to other blocks. Whenever the first statement of a block is executed. the other statements in the block are executed in the given order. Each block has statements for *definition* of variables and for *computational-uses* (c-uses. a variable is used in a computation or an output statement) of variables. An edge between block nodes represents a conditional or unconditional branch statement. An edge of a conditional branch is labeled with the condition. The variables used in the predicate of a condition are called *predicate-used* (p-use). A *path* is a finite sequence of connected nodes, $(n_1, ..., n_k)$, $k > 1$, such that there is an edge from n_i to n_{i+1}, for $i = 1, 2, ..., k - 1$. A path is *loop-free* if all nodes on the path are not in a cycle. A *definition-clear* path is defined as a path from a block containing a statement defining a variable to the block with a statement that uses the variable and where there are no other intermediate blocks with a statement redefining the variable along the definition-clear path. A *c-use path* denotes a definition-clear path from definition of a variable to c-use of that variable. A *p-use path* means a definition-clear path from definition of a variable to use of that variable in a predicate (condition). A *simple* path is a sequence of connected nodes, $(n_1, ..., n_k)$, $k > 1$, where all nodes are distinct except the first (n_1) and last (n_k). If all nodes are distinct, it is a *loop-free* path. A *du* path is a sequence of connected nodes, $(n_1, ..., n_{k-1}, n_k)$, $k > 1$, (1) if the last node (e.g., n_k) has a c-use of a variable, x, then the path is simple and definition-clear with respect to x, or (2) if the last edge (e.g., (n_{k-1}, n_k)) has a p-use of x and path $(n_1, ..., n_{k-1})$ is a definition-clear loop-free with respect to x.

Weyuker assumes the following constraints on a control flow graph representing a procedural program [RW85]:

(1) A control graph has exactly one start node (or block) which is the first node of the graph. The start node cannot be a destination of an edge.

(2) A control flow graph contains at least one halt node. The final node of the graph must be either a halt node or a node having an unconditional edge (branch) to some other node.

(3) A *syntactically endless loop* is defined as a circular path, $n_1, ..., n_k$, $k > 1$, $n_1 = n_k$, such that none of the nodes on the path contain either a conditional edge to a node which is not on the path or a halt node. Such a loop causes an infinite iteration. It is assumed that a control flow graph does not contain a syntactically endless loop [RW85]. This assumption assures that every node appears on some path from a start node to a halt node.

5.2.2 Procedural Testing Process

The process of testing procedural programs is shown in Figure 5.1. Assume that a set of testing requirements is given in which the program must perform. The testing process is described as follows:

(1) **Analyze Control Flow Graph**:

This involves three steps:

(a) The tester constructs a *node table*. There is a row for each node. The definitions and c-uses of variables in the nodes are entered in the node row.

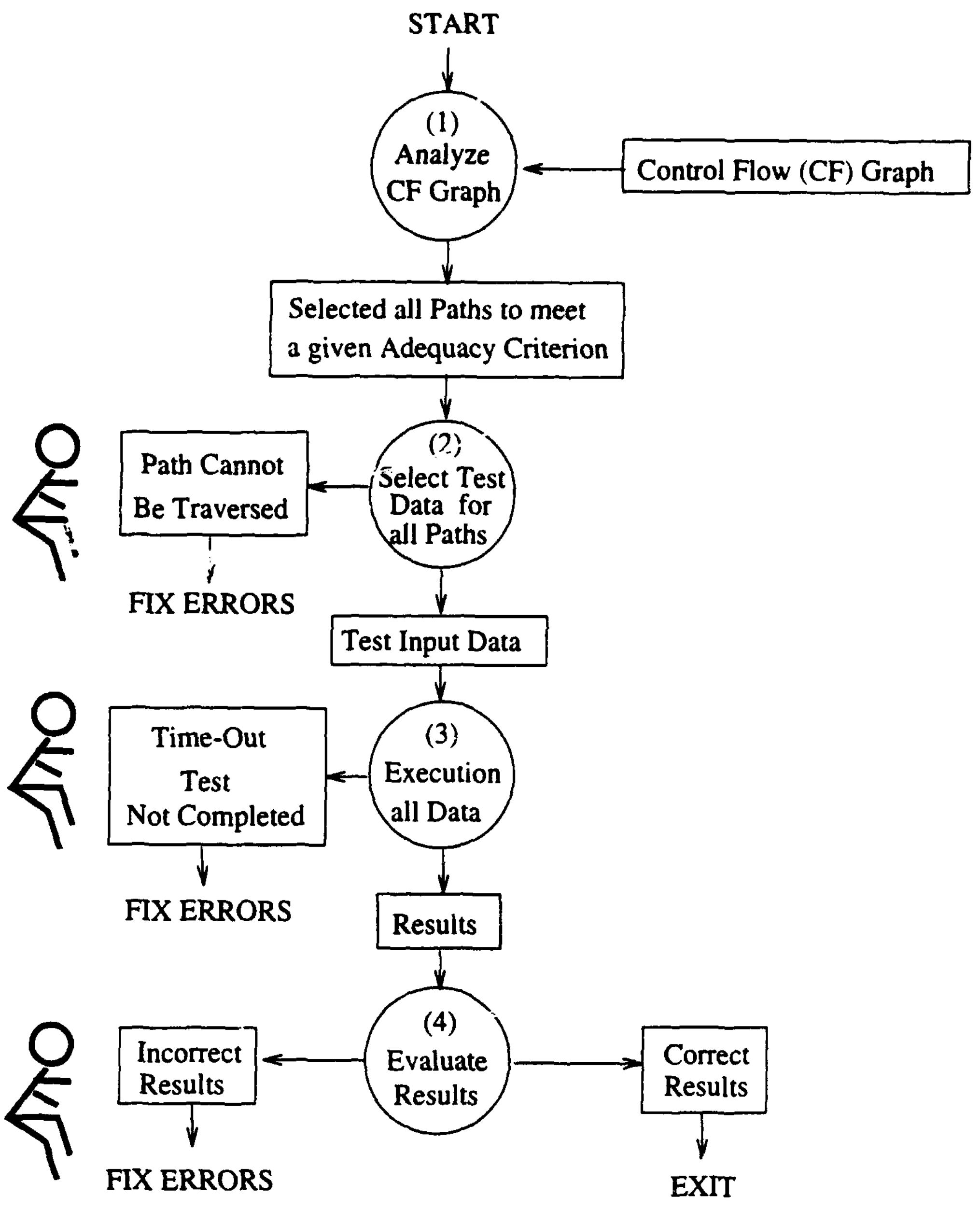


Figure 5.1: Procedural Testing Process.

An *edge table* is constructed. For each edge, there is a row. Respective set of p-use variables are listed for each edge.

(b) The tester finds all the definition-clear paths for each variable defined in each node. There are two types of definition-clear paths. The *definition-to-c-use paths* for a variable are obtained by finding the definition-clear paths that start from the node and end with c-uses of the variable. The *definition-to-p-use paths* are defined by finding definition-clear paths for a variable defined in the node that end with p-uses of the variable.

(c) The tester selects paths for testing to meet the specific adequacy criterion that are used. For example, the all-uses criterion requires testing of all the c-use paths and all the p-use paths for all variables (further discussed in Section 5.2.3).

(2) **Select Test Data**:

The tester finds test data which cause traversal of each definition-clear path selected. It is assumed that the tester can find input values which cause execution traversal of the selected path. This is an oracle assumption. These input variable values must satisfy instances when all the conditions of edges along the path are satisfied. Conditions along some paths may not be satisfiable. In that case, the human tester may also want to report to the developer to fix the errors causing the conditions unsatisfiable.

(3) **Execution**:

The program is executed with all the selected input values. The testing is performed for all paths selected to satisfy the adequacy criterion. If the test execution does not stop for a long time, the tester stops the execution. This is a time-out technique. The tester analyzes the program according the the requirements.

(4) **Evaluate the Results**:

Test results are evaluated by the tester. It is assumed that the tester will be able to determine whether or not the output values from a test execution are correct in the sense that they meet the requirements of the tested program. This may typically require independent calculations that reverse the program execution. This is an oracle assumption. If the results are incorrect, the tester reports the errors to the developer to make the necessary corrections. Otherwise, the testing process continues until the test adequacy criterion is satisfied.

5.2.3 Procedural Adequacy Criteria

This section concerns the procedural testing adequacy criterion used in step (1) of the process of Figure 5.1. The notions of the paths used in the Procedural Testing are the basis of defining adequacy criteria [Bei90, Wey90].

The criteria can be related in terms of a transitive relation called *inclusion*, $\supset$, which denotes relative "strength" of two criteria [RW85]. A criterion, R_i, is defined to be "*stronger*" than a criterion, R_j, if all test cases produced under R_j are included in those produced under R_i, i.e., $R_i \supset R_j$. Figure 5.2 illustrates a hierarchy of strength relations among criteria [RW85]. The criteria are as follows:

- *all-paths*: Test input data must cause traversal of every path from the start node to the halt node of the control flow graph. This is the strongest criterion. Each test data must satisfy combinations of conditions of the edges along paths from a start node to a halt node. The paths may include cycles. A cycle must be traversed at least once in respective iterations but different paths may be traversed in different iterations. As noted, a cycle with infinite iterations cannot make the iteration finish. Using a timeout technique, a cycle that is iterated for a long time may be detected and then analyzed to determine the cause.

- *all-du-paths*: Test input data must cause traversal of every du path from every

$$\begin{tikzcd}
 & & \text{all-paths} \arrow[d] & & \\
 & & \text{all-du-paths} \arrow[d] & & \\
 & & \text{all-uses} \arrow[dl] \arrow[dr] & & \\
 & \text{all-c-uses/ some-p-uses} \arrow[dl] \arrow[dr] & & \text{all-p-uses/ some c-uses} \arrow[dl] \arrow[dr] & \\
\text{all-c-uses} & & \text{all-definitions} & & \text{all-p-uses} \arrow[d] \\
 & & & & \text{branch} \arrow[d] \\
 & & & & \text{statement}
\end{tikzcd}$$

Figure 5.2: Hierarchy of adequacy criteria in procedural programming.

definition of every variable to every use of that definition. The all-du-paths criterion is weaker but more practical than the all-paths criterion: (a) it includes paths which are subpaths of those required in the all-paths criterion. (b) it excludes paths including loops.

- *all-uses*: Test data must cause traversal of at least one path from every variable definition to every use of that variable. There can be several du paths from a variable definition node to a respective use node (c-use) or a respective use edge (p-use). The all-uses criterion requires the traversal of only one of such paths, while the all-du-paths criterion requires traversal of all such paths.

- *all-c-uses/some-p-uses*: Test data must cause traversal of at least one definition-clear path from each variable definition to every c-use of that variable. If any definition is not used in a c-use but used only in a p-use. add such p-use paths. This will assure that every definition is included in some test. It might well miss an error caused by omitting some p-use paths.

- *all-p-uses/some-c-uses*: Test data must cause traversal of at least one definition-clear path from each variable definition to every p-use of that variable. If a definition is not c-used, add such c-use paths. This assures that every definition is included in some test. It could miss an error caused by omitting some c-use paths.

- *all-c-uses*: Test data must cause traversal of at least one definition-clear path from each variable definition to every c-use of that variable.

- *all-definitions*: Test data must cause traversal of at least one definition-clear path from each variable definition to some p-use or some c-use.

- *all-p-uses*: Test data must cause traversal of at least one definition-clear path from each variable definition to every p-use of that variable definition.

- *branch* (or *all-edges*): Test data must cause traversal of each edge of a control flow graph. This requires test data that satisfies the conditions on the respective edges, but not conjunction of the conditions along a path.
- *statement* (or *all-nodes*): Test data must cause traversal of each node of a definition use graph.

5.3 Equational Visual Testing Process

Figure 5.3 illustrates the Equational Visual Testing process. It is similar to the process of the Procedural Testing in Figure 5.1 except that it includes an additional step for recompilation for testing. These steps are briefly described below and then discussed further in later sections.

(1) **Compile for Testing**:

An equational specification is modified (possibly automatically) and recompiled before it is tested. The modified program execution stores the values of interim variables and facilitates tracing exclusive paths from input nodes to output nodes. Modifications for testing consist of the following:

(a) Equations with multiple conditional expressions (*complex conditional equations*, e.g., nested IF-THEN-ELSE) are expanded each to a set of single conditional equations, each with a respective expression (a *simple conditional equation*, e.g., IF-THEN). The conditions in the set of simple conditional equations must be *exclusive*. Namely, only one of them can be satisfied for any input. Equations are also added to define explicitly the conditions in the simple conditional equations.

(b) Values of interim and control variables are declared as target variables (See Chapter 2). This amounts to declaring these variables as output on a secondary storage media.

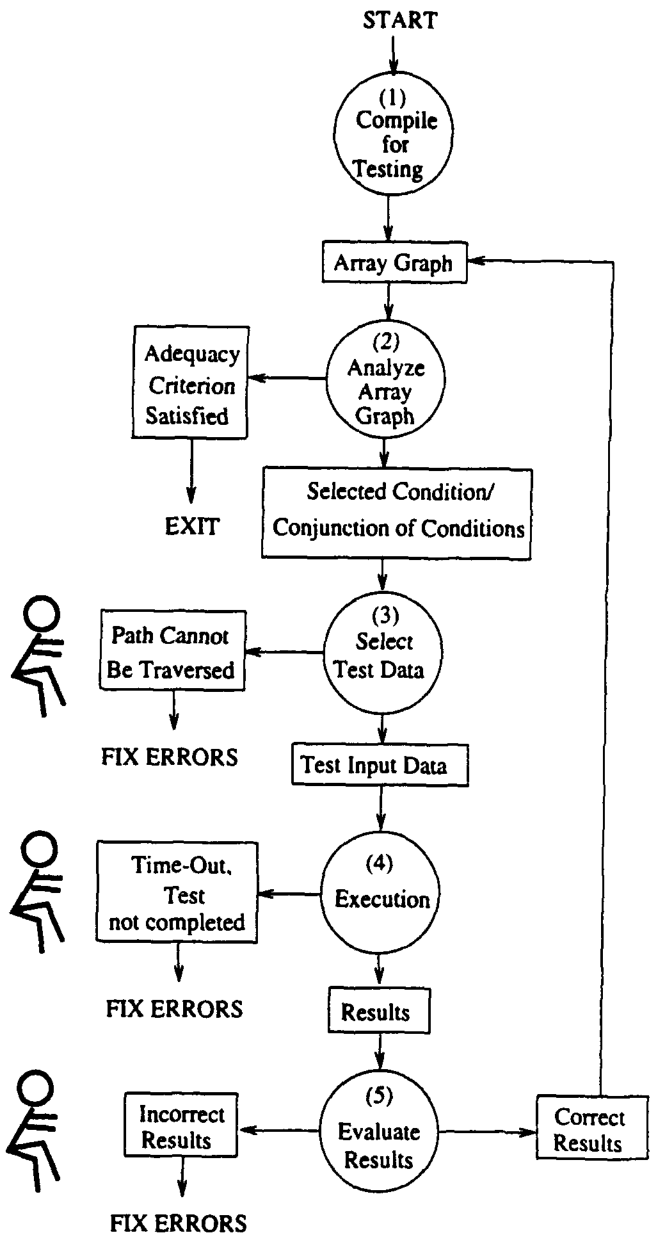


Figure 5.3: Equational Visual Testing Process.

The modified equational specification is then compiled for the testing. An array graph is generated for the visualization. Executable code is generated and used for executing the tests.

(2) **Analyze Array Graph and Select a Test Condition**:

An array graph of the modified equational specification is then shown on the screen. Note that the conditions of the simple equations expanded from a complex conditional equation must be exclusive. Conditions are selected from the exclusive set, one by one. Next, input values are selected to satisfy the condition. The equational specification in then executed with the test input values. The nodes of the conditions, which have been previously selected and tested, are cumulatively shaded. The more conditions are satisfied during tests. the more condition nodes are shaded in the graph. The shaded condition nodes visually denote the progress in test coverage. The testing process ends when the test adequacy criterion is satisfied. This may mean that all conditions required by the adequacy criterion are shaded. The adequacy criterion may require one or multiple test input values that satisfy each condition. Note also that each test input value set may cause the shading of more than one condition node.

(3) **Select Test Data**:

Test input values are determined to satisfy each selected condition in step (2). In some adequacy criteria, combination of conditions must be satisfied. The input data variables are obtained by backtracking, along the causality paths, from the condition to the respective input variables. The test input values are chosen from the domains of the input variables that satisfy the condition. Determining values of input variables that satisfy the condition may be complex. It is further discussed in Section 5.6.3. This is a human operation which is assumed to be able to find the values of the input variables satisfying the conditions. This is an "Oracle" assumption [Bei90]. If the conditions are not satisfiable, then the tester reports the developer to fix the problem. The input data are entered by pointing to the nodes of the associated input variables of the array graph. Respective files for input variable values are then created.

(4) **Execution**:

The compiled specification is executed with each selected test input values, shading condition nodes and producing the test results. The output values from the test execution can be viewed via pointing to their corresponding variable nodes. Note that a time-out is used to terminate the test if execution time is unreasonably long.

(5) **Evaluate Results**:

It is assumed that the human tester will be able to determine whether or not the output values from a test are correct. This is further discussed in Section 5.6.5. This is also an oracle assumption [RW85, Bei90]. If the results are incorrect, the tester reports the errors to the developer so that the equational specification can be fixed and then testing is restarted. Otherwise, the testing process continues from step (2) until the test adequacy criterion is satisfied.

5.4 Pre-Test Compilation

The objective of pre-test compilation is to modify the array graph through a compilation to represent conditions and respective simple conditional equations as nodes in the array graph. This is step (1) in Figure 5.3. The pre-test compilation aims (a) to use only unconditional or simple equations (with exclusive conditions) (b) to store away values of all interim variables. The pre-test compilation expands a complex conditional equation into a set of simple conditional equations. The process of expansion is similar to the code optimization technique used in an equational compiler [Bru89].

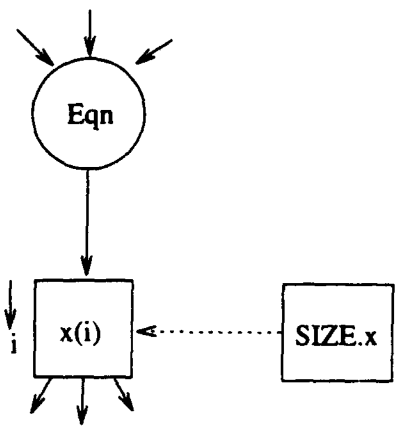


Figure 5.4: An equation before the condition expansion.

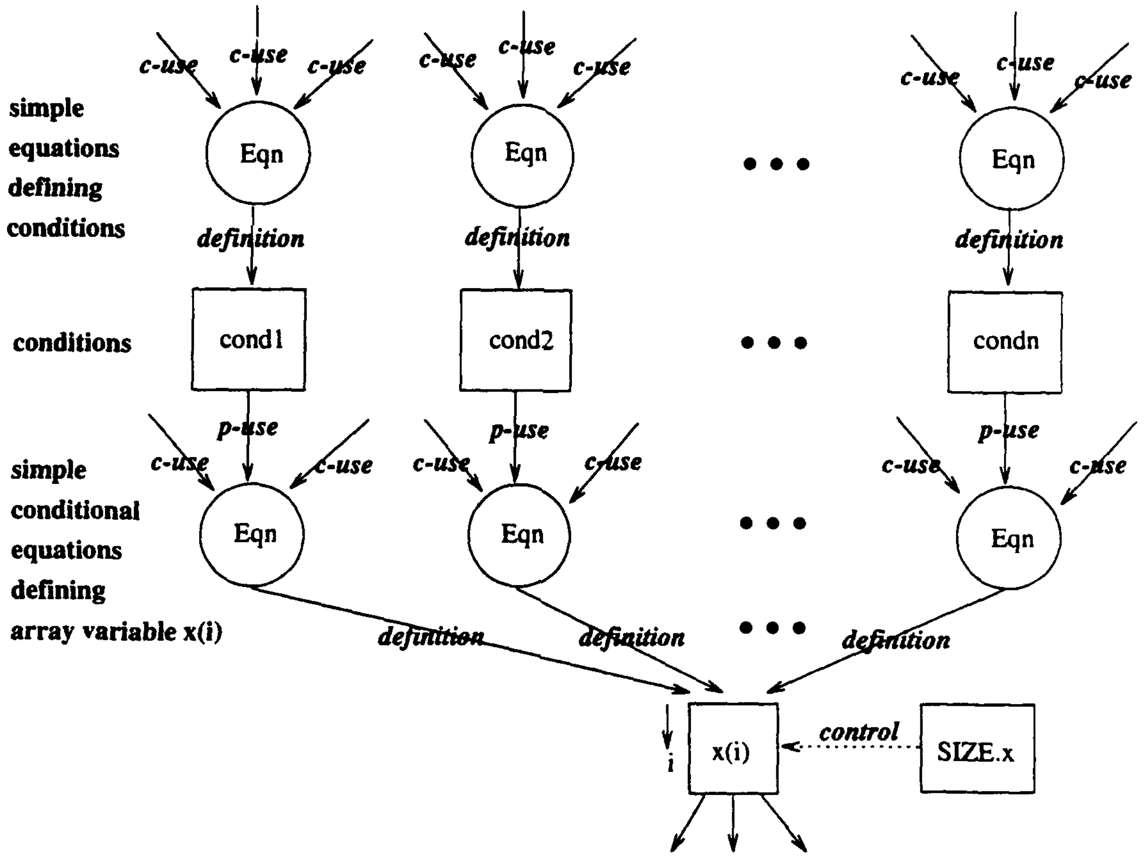

simple equations defining conditions
c-use
c-use
c-use
Eqn
definition
conditions
cond1
cond2
condn
p-use
simple conditional equations defining array variable x(i)
Eqn
definition
i
x(i)
control
SIZE.x

Figure 5.5: The complex conditional expressions are expanded and the edges are marked as definition, c-use, p-use, and control.

An equation is either non-conditional, simple conditional (with a single conditional expression, e.g., IF-THEN), or complex conditional (with more than one conditional expressions, e.g, nested IF-THEN-ELSE). A non-conditional equation is activated as soon as all of its input data are available. A simple conditional equation is activated when the condition is satisfied and the input data are available. Therefore, the conditional expression of the simple conditional equation becomes the condition of executing the equation. A complex conditional equation is expanded into a series of simple conditional equations of which the conditions are exclusive. For instance, let the complex conditional equation in Figure 5.4 (the control variable. SIZE.x, defines the size of the array variable) be:

```
x(i) = IF c1 THEN exp1
                 ELSE IF c2 THEN exp2
                             ELSE IF c3 THEN exp3
                                         ELSE IF ...
                                          ...
                                         ELSE IF cn1 THEN expn1
                                                     ELSE expn;
```

It is expanded into a series of simple equations shown below. First, variables. cond1, cond2,..., condn, are introduced for the conditions and defined using the conditional expressions, c1, c2,..., cn, by simple equations as follows:

```
cond1 = c1;
cond2 = ~c1 & c2;
cond3 = ~c1 & ~c2 & c3;
 ...        ...        ...
condn = ~c1 & ~c2 & ... & ~cn1;
```

The equational compiler must assure that a condition variable is not duplicated in the array graph. Whenever a new condition variable is introduced, the compiler compares it with existing condition variables. The simplest method is to compare two conditional expressions using string matching. A more powerful method would use a term rewriting system, which transforms every conditional expression into a normal form, for the expression comparison. The string matching method is used in the compiler because it is easier to implement.

The complex conditional equation is then split into simple conditional equations using the conditions as below:

```
x(i) = IF cond1 THEN exp1;
x(i) = IF cond2 THEN exp2;
x(i) = IF cond3 THEN exp3;
   ...          ...
x(i) = IF condn THEN expn;
```

The process of the condition expansion is applied to each complex conditional equation in the array graph until every complex condition is expanded.

Finally, all interim variables are listed as target variables (See Section 2.3). These variables are thus defined as members of target records stored in secondary storage.

5.5 Adequacy Criteria for Equational Visual Testing

It is proposed to use **equational all-paths, equational all-du-paths, equational all-uses**, and **equational all-definitions** adequacy criteria in the Equational Visual Testing. They are partly similar to the respective criteria in the Procedural Testing. The equational test adequacy criteria are defined in this section.

The Equational Visual Testing is based on the adequacy criteria specified in terms of paths. A path in an array graph is a sequence of nodes connected by edges. The different criteria will require test execution traversal of specific types of paths as discussed later.

Recall that edges in an array graph denote the following.

(1) **data dependency** between a variable node and an equation node or vice versa: If an edge is from a variable node to an equation node, it means the variable is used in the RHS (right hand side) of the equation. An edge from the equation node to the variable node denotes a variable defined by the equation.

(2) **parameter dependency** between a control variable and its associated array variables: A control edge from a control variable such as `SIZE` and `END` to an array variable defines either a number of iterations for computing the array variable (`SIZE`) or a condition for terminating the computation (`END`).

(3) **hierarchical dependency** between files, records, groups, and fields: The edges denoting hierarchical dependencies are always traversed during test execution.

A single-edge path in the Equational Visual Testing can be characterized as *definition*, *p-use*, *c-use*, and *control*. The single-edge paths are illustrated in Figure 5.5. The paths are defined as follows:

(1) A **definition** path is an edge from an equation node to its LHS (left hand side) variable node. The LHS variable can be a condition. The edges such as (Eqn, cond1), (Eqn, cond2),..., and (Eqn, condn) in Figure 5.5 are definition paths of the conditions, cond1, cond2,..., and condn. The edge of (Eqn, `x(i)`) in Figure 5.5 is a definition path of the variable, `x(i)`.

(2) A **p-use** path is an edge from a condition variable node to a simple conditional equation node that references the condition variable (the simple conditional equations defining `x(i)` in Figure 5.5) such as (cond1, Eqn), (cond2, Eqn),...,(condn, Eqn).

(3) A **c-use** path is an edge from a non-conditional variable node to an equation node that references the variable in its RHS expression, e.g., (c1, Eqn), (c2, Eqn),...,(cn1, Eqn), in Figure 5.5.

(4) A **control** path is an edge from a control variable to the corresponding array variable, e.g., (`SIZE.x`, `x(i)`) in Figure 5.5.

5.5.1 The equational all-paths criterion

This criterion is the strongest and demands the most complex analysis to select and verify tests.

Definition

The equational all-paths criterion requires test execution traversal, at least once, of each existing path from anyone of the source nodes to anyone of the target nodes in the array graph.

If the array graph contains MSCC's, then edges are deleted to "open" the MSCC's and obtain an acyclic array graph. The selection of edges to be deleted was discussed in Chapter 4. The deleted edges are labeled with a subscript expression of the form, `I - K`, where `I` is a subscript and `K` is a positive integer. Such edges are deleted from an MSCC, one by one, until no more MSCC is in the graph. A deleted edge has a node at its inception and a node at its termination. The node at the inception of the deleted edge is also considered as a target node and the node at the termination of the deleted edge is also considered as a source node, for the purpose of defining paths for the equational all-paths criterion. The order of deleting edges may affect the paths from source to target. In that case, all the different orders of deleting edges must be investigated to define the source-to-target paths properly.

Thus the acyclic array graph has a finite number of source-to-target paths to be traversed for satisfying the equational all-paths criterion.

Testing Process

The problem with the equational all-paths criterion is that it requires considering many tests. To satisfy the traversal of a path, test input values must satisfy

Test No.	Conditions	Mark
1	cond 11, cond 21,..., cond m1	$\surd$
2	cond 11, cond 22,..., cond m1	
...	...	...
j	cond 11, cond $2n_2$,..., cond mn_m	$\surd$
j+1	cond 12, cond 21,..., cond m1	X
...	...	...

Table 5.2: A condition table

conjunctions of conditions along the path. A path may have none, one or several condition nodes. Some paths may have the same conditions. Some paths are never traversed because the conjunctions of conditions along the paths may not be satisfiable. Handling these cases will be discussed below. The following procedure reduces the average number of test executions and especially the labor of finding test input values that satisfy the conjunctions of the condition variables along a respective path.

The Equational Visual Testing system can traverse each path and find the conditions along the path that must be satisfied conjunctively. The conditions along the paths can be listed in a *condition table*, illustrated in Table 5.2, where $\surd$ means that the path is traversed and X means that the path is not traversable. Rows with duplicate conjunctions of conditions are omitted in the table. The human tester may select a yet unsatisfied (unshaded, explained later) single condition and find test input values that satisfy the condition. Next, the test is executed with selected test input values that satisfy the single condition. The system can examine all the evaluated values of conditions and find the paths in which all conditions were satisfied during the test execution. It then marks the respective rows in the condition table. It also shades the condition nodes along the marked paths. Thus, more than one condition nodes may be shaded in the array graph and more than one paths may be marked due to a single text execution, especially if there are array conditions in the paths that are satisfied for different element subscript values.

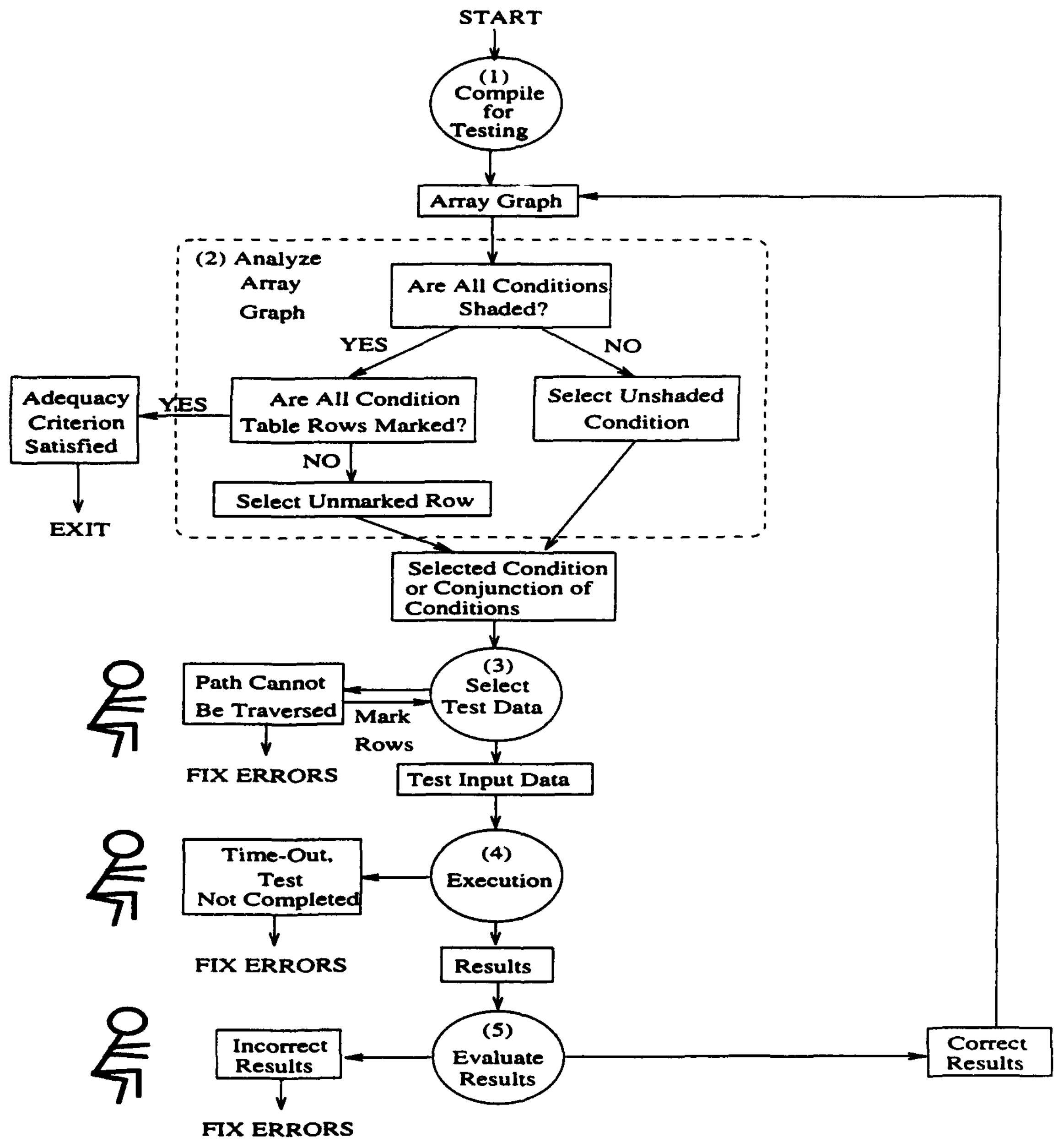


Figure 5.6: Process of satisfying the all-paths criterion.

The process of satisfying the equational all-paths criterion is shown in the process chart in Figure 5.6 which is identical to Figure 5.3 except the exploded node of step (2), "Analyze Array Graph". After the pre-test compilation in step (1), the human tester checks the array graph if all the conditions are shaded (satisfied). If not, the tester selects a yet unshaded (unsatisfied) condition and goes to step (3). If all condition nodes are shaded, the tester examines the condition table to check if there is an untested path in the array graph. If all rows of the condition table are marked, it means that all paths are traversed during the previous test executions. That is, the equational all-paths criterion is satisfied. Otherwise, a yet unmarked row (a conjunction of conditions) is selected for the testing. This is step (2). In step (3), the selected condition or conjunction of conditions are examined. A single condition must be satisfiable. Otherwise, it must be erroneously composed. This should be reported to the developers so that the errors can be fixed. If the single condition is satisfiable, the tester finds test input values to satisfy the selected condition (possibly for some subscript value). Suppose conjunction of conditions is selected in step (2). The selected conjunction may not be satisfiable and still this does not mean that this is an error. Then, the tester marks the row with X meaning that the conjunction is not satisfiable, i.e., the associated path is not traversable. This must be investigated by the human tester or developer to find the reason why the path is not traversable. If the conjunction is satisfiable, the tester finds test input values to satisfy the selected conjunction. Next, the test execution is exercised with the selected test input values in step (4). The conditions satisfied during the test execution are shaded. The rows of the condition table are marked according to the results of the test execution. This is step (4). Step (5) and the rest of the testing process are as same as described in Section 5.3. This procedure is repeated until every condition is satisfied and all the rows of the condition table are marked (the equational all-paths criterion is satisfied).

This procedure reduces both the number of test executions and the labor in finding test input values for each test. It is further illustrated in Section 5.6.

Comparison

The procedural all-paths criterion is impractical for testing procedural programs because of a start-to-end path in a control flow graph that includes cycles (denoting iteration loops) [RW85]. It is impossible to define how many times such a start-to-end path in a control flow graph should traverse cycles. Even if it is required to traverse all nodes in the graph only once, the problem is still impractical. That is, the problem becomes the directed Hamiltonian circuit problem which is an *NP-complete* problem [AHU74, Eve79, Bol79]. There is a general agreement that an NP-complete problem is intractable. Thus it is impractical, in general, to define paths for the procedural all-paths criterion.

If we could obtain an acyclic graph from the control flow graph, the problem of defining all-paths would become practical. That is, it becomes a problem of finding the *spanning tree* of an acyclic graph of which worst-case time complexity is known to be $O(e)$, where e is the number of edges in the graph [AHU74]. However, it is not possible to make the control flow graph acyclic. Note that edges of the control flow graph cannot be deleted because they denote branch statements. If they are deleted, the control structure of the program is destroyed.

On the other hand, in the equational all-paths criterion, every MSCC of the array graph can be opened to obtain an acyclic graph by deleting edges (as discussed in Chapter 4). The process is performed based on subscript expressions of edges during the compilation. The deletion of an edge from an MSCC means that the computation of the variables in the MSCC can be exercised without any conflict in data dependency between the variables. If there is an unopenable MSCC, the equational compiler generates an error message. Thus the array graph becomes acyclic in performing compilation so that the program behavior is not affected by the deleted edges. The tester can also use the acyclic graph to find all paths that can be traversed. A condition table is generated for the equational all-paths criterion.

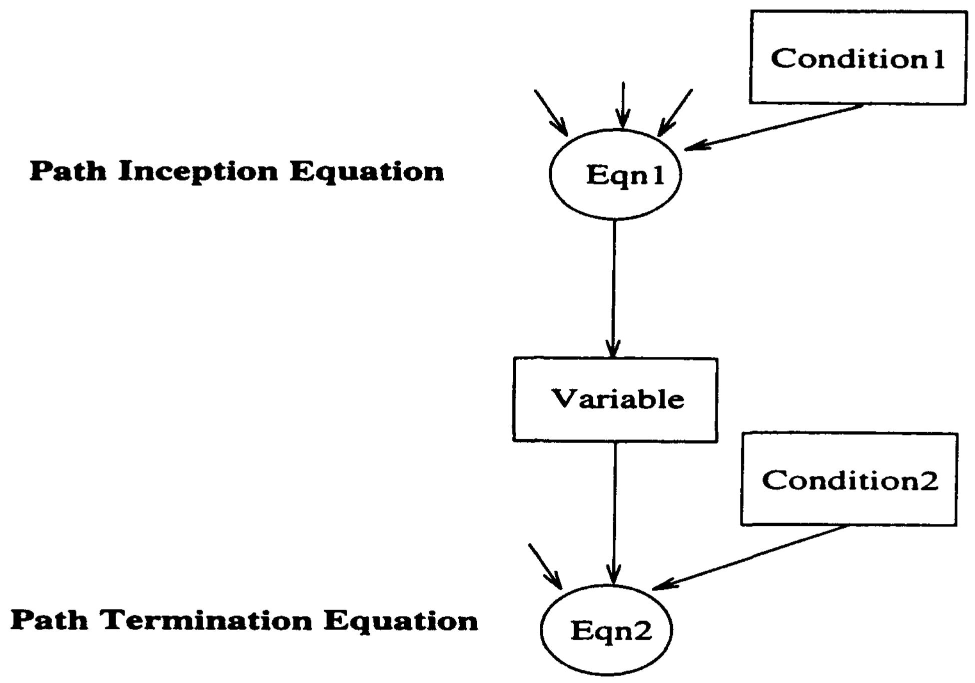


Figure 5.7: A du-path in an array graph.

5.5.2 The equational all-du-paths criterion

This criterion applies to the subpaths of the equational all-paths criterion. Therefore. it is less severe because it does not require traversing all the possible combinations of subpaths.

Definition

The equational all-du-paths criterion requires traversal of every path from a variable definition (an equation node defining a variable) to all the equations that use the variable (an equation node using the variable). Namely every du-path in an array graph. A du-path in the array graph is a two edge path from an equation node to another equation node. This is illustrated in Figure 5.7 and also further exemplified in Section 5.6.3. Note that it is not required to open MSCC's in the array graph for this test, in order to define a finite number of paths, as was needed for the equational all-paths criterion.

Testing Process

As shown in Figure 5.7, traversal of a du path can be selected by applying test input values that satisfy at most a conjunction of the two conditions that are inputs to the equations: a condition for the variable defining equation (such as `Condition1` in Figure 5.7) and a condition for the variable using equation (such as `Condition2` in Figure 5.7). There can be at most one condition that is an input to an equation (see Section 5.4). Again, the Equational Visual Testing system can find all the paths as defined above in order to construct the condition table. Note that it is not necessary to open MSCC's. Paths which can be selected by satisfying one condition or a conjunction of two conditions are placed in the condition table. Paths which do not have any conditions may be omitted, because they will be traversed on every test execution. The process of Figure 5.6 is followed here as well.

Comparison

The procedural all-du-paths criterion [RW85] is not as complete as the equational all-du-paths criterion because it omits testing some cyclic paths (all except a single edge cycle from definition to c-use). A path from definition to use in the control flow graph may be longer in number of edges and require satisfying conjunctively more than two conditions to select traversal of the path. Each variable is an inception node of all of its du-paths. In the procedural testing, a du-paths may consist of multiple edges. There can be multiple nodes that are inception nodes of du-paths for the variable. Thus, it is relatively easy in the Equational Visual Testing to construct the condition table ($O(n)$, where n is a number of variable nodes in an array graph) and exercise the testing process for satisfying the equational all-du-paths criterion.

5.5.3 The equational all-uses criterion

This criterion requires traversal of the subpaths of the equational all-du-paths criterion. It involves subpaths of the previous criterion from a variable to its uses.

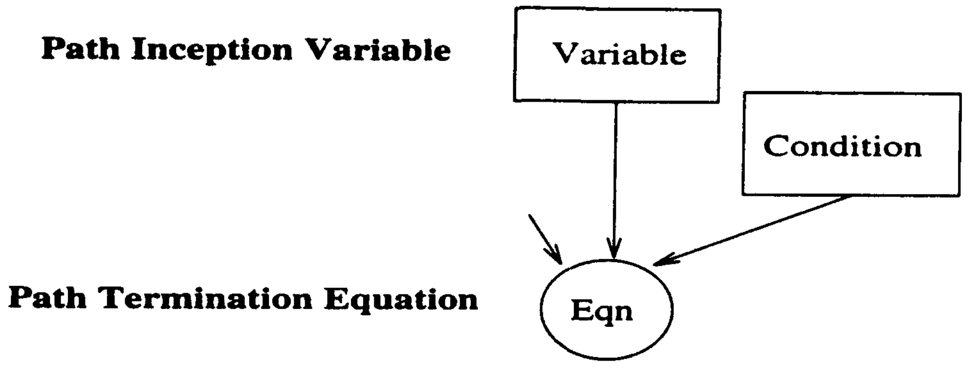


Figure 5.8: A use path in an array graph.

Definition

The equational all-uses criterion requires traversal of at least one path from every variable node to every equation node which uses that variable. The paths for the criterion are single-edge paths from a variable node to an equation node. Such a path is shown in Figure 5.8. It is not required to construct the condition table as was for the equational all-paths and the equational all-du-paths criteria. This is because no conjunction of conditions is involved.

Testing Process

The equational all-uses criterion requires to select test input values for each unshaded condition in the array graph. The tester selects an unshaded single condition and finds test input values to satisfy the selected condition. The test execution is performed with the test input values. Condition nodes are shaded according to the results. The procedure is repeated until all the condition nodes are shaded.

Comparison

The equational all-uses criterion has a simple definition of a path which is represented by a single edge. In the procedural all-uses criterion, the path may consist of multiple edges. The testing process can be completed by selecting a condition node, one by one, and finding test input values to satisfy the selected condition. This reduces the labor of finding test input values and evaluating test results.

5.5.4 The equational all-definitions criterion

This is the weakest criterion.

Definition

The equational all-definitions criterion requires traversal of one definition path of every variable. It does not require to test all exclusive definitions of variables. A single execution satisfies this criterion.

Testing Process

The process of satisfying the equational all-definitions criterion is trivial. Since every variable is defined in an equational specification, one execution of the equational specification satisfies the equational all-definitions criterion.

Comparison

As discussed, it is trivial to satisfy the equational all-definitions criterion. But the procedural all-definitions criterion requires to traverse multiple assignment statements which define and update variables.

5.6 Example

This section illustrates the Equational Visual Testing through an example of testing the MODEL specification of the Euclid algorithm for finding GCD of two integers (presented in Section 2.2). The testing process follows the chart in Figure 5.3.

```
/* EQUATIONS */

/* conditions for expanded equations */
/* Eq c-1 */ cond1 = (i=1) & x1>x2;
/* Eq c-2 */ cond2 = (i=1) & ^(x1>x2);
/* Eq c-3 */ cond3(i) = ^(i=1) & (y1(i-1)>y2(i-1));
/* Eq c-4 */ cond4(i) = ^(i=1) & ^(y1(i-1)>y2(i-1));

/* expanded equations of Eq 1: define values of y1(i) */
/* Eq 1-1 */ y1(i) = IF cond1 THEN x1;
/* Eq 1-2 */ y1(i) = IF cond2 THEN x2;
/* Eq 1-3 */ y1(i) = IF cond3(i) THEN y1(i-1)-y2(i-1);
/* Eq 1-4 */ y1(i) = IF cond4(i) THEN y1(i-1);

/* expanded equations of Eq 2: define values of y2(i) */
/* Eq 2-1 */ y2(i) = IF cond1 THEN x2;
/* Eq 2-2 */ y2(i) = IF cond2 THEN x1;
/* Eq 2-3 */ y2(i) = IF cond3(i) THEN y2(i-1);
/* Eq 2-4 */ y2(i) = IF cond4(i) THEN y2(i-1) - y1(i-1);

/* Eq 3: determines the terminating condition (not expanded) */
END.y1(i) = (y1(i) = y2(i));

/* Eq 4: computes z=GCD(x1,x2) (not expanded) */
z=IF END.y1(i) THEN y1(i);
```

Figure 5.9: The expanded equational specification for the GCD example.

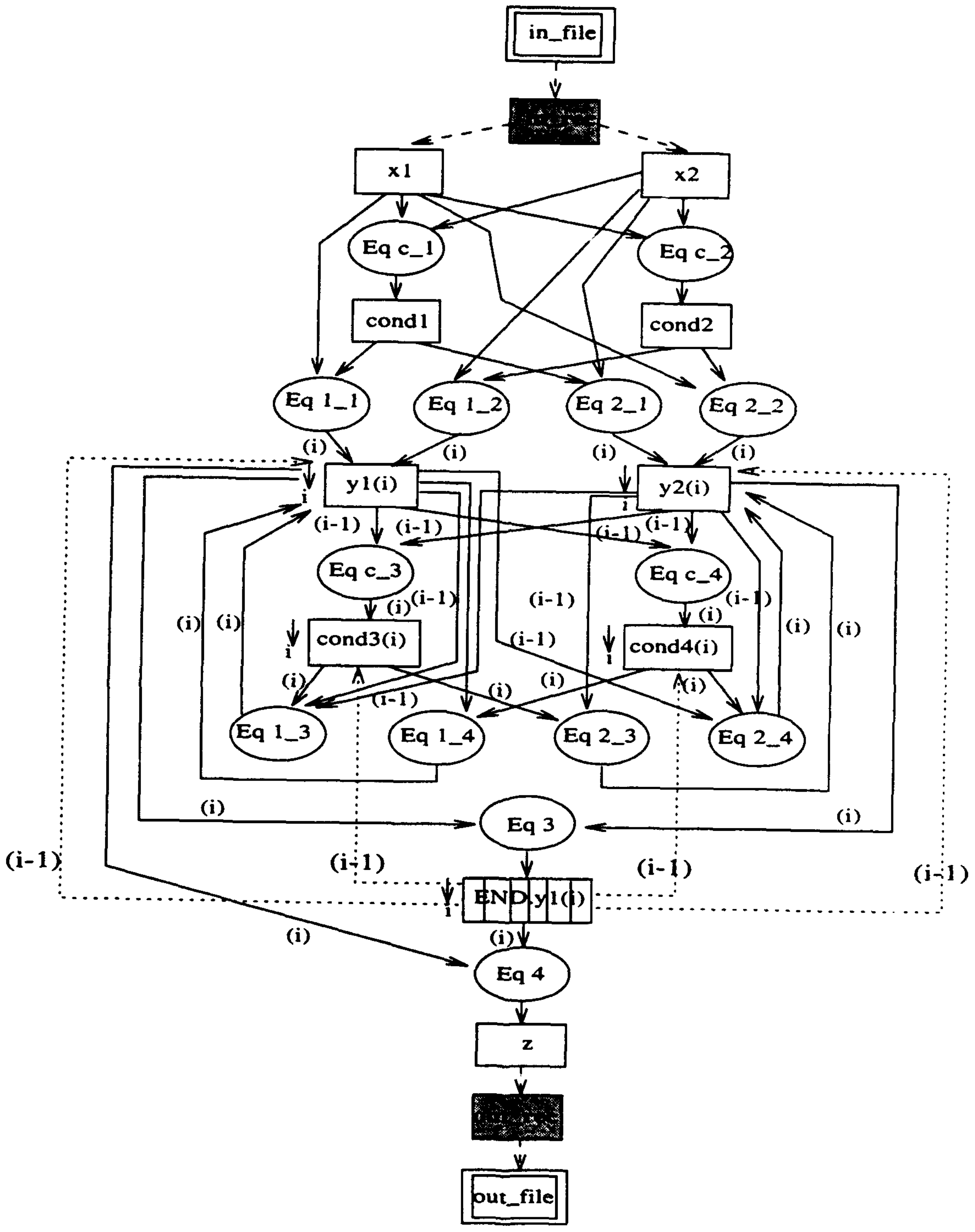


Figure 5.10: Array graph of the GCD example.

Deleted Edge (ordered)	Source	Target
(END.y1(i), y1(i))	y1(i)	END.y1(i)
(END.y1(i), y2(i))	y2(i)	END.y1(i)
(END.y1(i), cond3(i))	cond3(i)	END.y1(i)
(END.y1(i), cond4(i))	cond4(i)	END.y1(i)
(y1(i), Eq c-3)	Eq c-3	y1(i)
(y1(i), Eq c-4)	Eq c-4	y1(i)
(y1(i), Eq 1-3)	Eq 1-3	y1(i)
(y1(i), Eq 1-4)	Eq 1-4	y1(i)
(y1(i), Eq 2-4)	Eq 2-4	y1(i)
(y2(i), Eq c-3)	Eq c-3	y2(i)
(y2(i), Eq c-4)	Eq c-4	y2(i)
(y2(i), Eq 2-3)	Eq 2-3	y2(i)
(y2(i), Eq 2-4)	Eq 2-4	y2(i)

Table 5.3: Deleted edges to open MSCC's and new sources and targets.

5.6.1 Pre-Test Compilation (step(1))

The GCD specification is recompiled for testing as discussed in Section 5.4. The complex conditional equations are transformed into simple conditional equations. The compiler also creates equations for new condition variables, `cond1`, `cond2`, `cond3(i)`, and `cond4(i)`. The expanded equational specification is presented in Figure 5.9.

5.6.2 Analyze Array Graph (step (2))

The array graph for the recompiled equational specification is visualized in Figure 5.10. The analysis of the array graph depends on the adequacy criterion used.

The Equational All-Paths Criterion

The MSCC's in the array graph are opened to obtain an acyclic graph (see Chapter 4 and Section 5.4). Edges are deleted, one by one, until all the MSCC's in the array graph are "opened". Note that there are a number of sequences in deleting edges. The deleted edges in one of the sequences are listed in Table 5.3. The inception nodes and the termination nodes of the deleted edges become new target and source

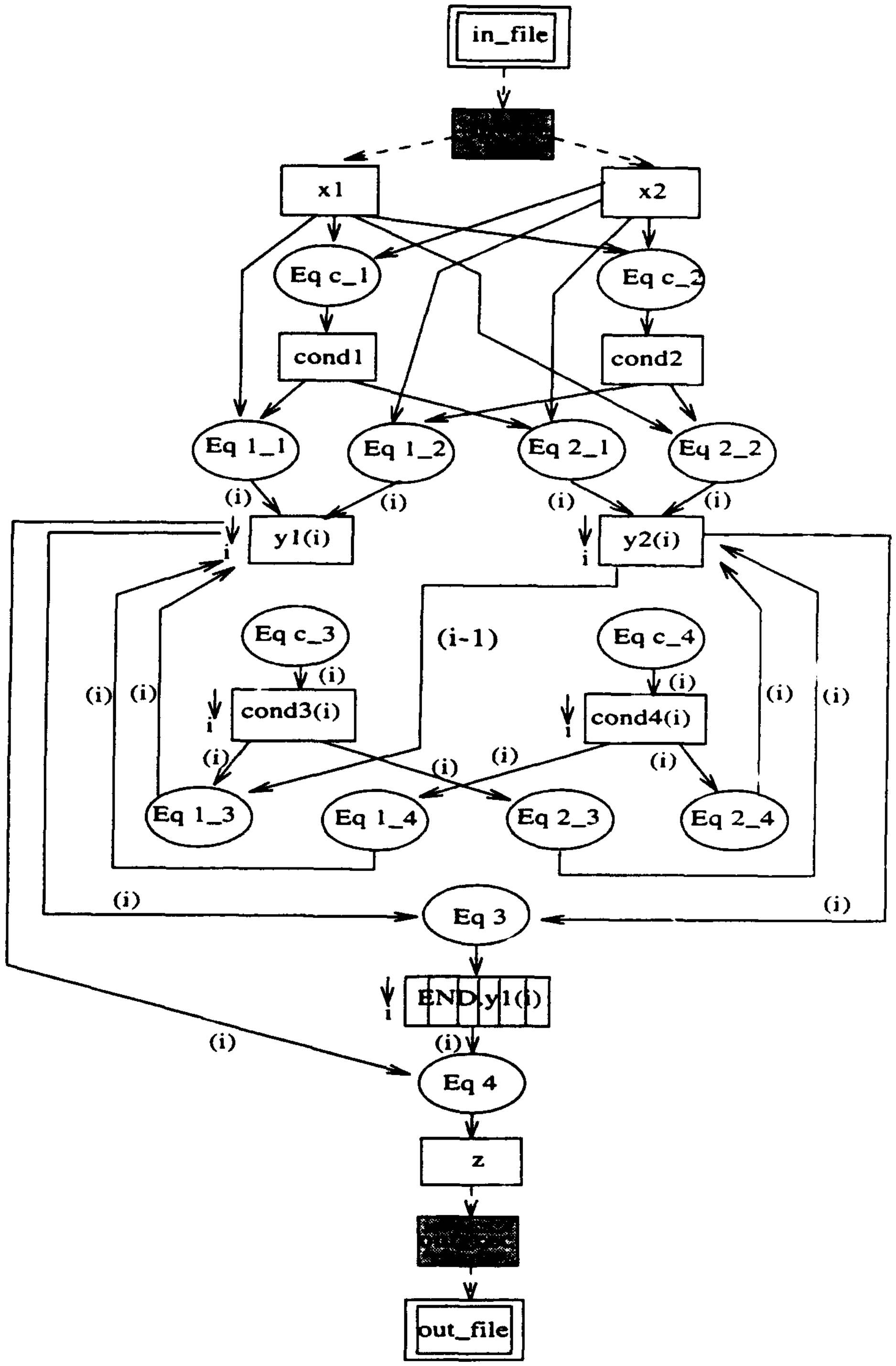


Figure 5.11: Acyclic array graph of the expanded GCD example.

No.	Conditions	Mark
1	cond1	
2	cond2	
3	cond3(i)	
4	cond4(i)	
5	cond1, cond3(i)	
6	cond2, cond3(i)	
7	cond1, cond4(i)	
8	cond2, cond4(i)	
9	cond1, END.y1(i)	
10	cond2, END.y1(i)	
11	cond3(i-1), cond4(i)	
12	cond4(i-1), cond3(i)	
13	cond3(i-1), END.y1(i)	
14	cond4(i-1), END.y1(i)	
15	cond1, cond3(i-1), END.y1(i)	
16	cond2, cond3(i-1), END.y1(i)	
17	cond1, cond4(i-1), END.y1(i)	
18	cond2, cond4(i-1), END.y1(i)	
19	cond3(i-2), cond4(i-1), END.y1(i)	
20	cond4(i-2), cond3(i-1), END.y1(i)	

Table 5.4: Condition table for the equational all-paths criterion.

No.	Conditions	Mark
1	cond1	
2	cond2	
3	cond3(i)	
4	cond4(i)	
5	cond1, cond3(i)	
6	cond2, cond3(i)	
7	cond1, cond4(i)	
8	cond2, cond4(i)	
9	cond1, END.y1(i)	
10	cond2, END.y1(i)	
11	cond3(i-1), cond4(i)	
12	cond4(i-1), cond3(i)	
13	cond3(i-1), END.y1(i)	
14	cond4(i-1), END.y1(i)	

Table 5.5: Condition table for the equational all-du-paths criterion.

nodes of the corresponding acyclic array graph, respectively. These are also listed in Table 5.3. Figure 5.11 illustrates the acyclic array graph for the expanded GCD example. The source nodes of the acyclic array graph are x1, x2, and the source nodes listed in Table 5.3. The target nodes of the acyclic graph are z and the target nodes listed in Table 5.3. The tests must traverse all source-to-target paths of the acyclic array graph. As discussed in Section 5.5, the Equational Visual Testing system generates a condition table of the conjunctions of the conditions in respective paths. The table reflects the different possible order of deleted edges. These are shown in Table 5.4. Note that multiple paths containing the same condition variables are merged into one row in the table.

The Equational All-Du-Paths Criterion

It is not required to open MSCC's of the array graph for this criterion. Table 5.5 presents the condition table for the criterion. Note that paths without conditions are not included as they are traversed in every test execution. Also note that the rows in Table 5.5 are equal to or subset of the rows in Table 5.4.

The Equational All-Uses Criterion

The tester selects single conditions, one by one. The test is then applied (step 3, 4, and 5). This repeats until all conditions have been shaded. It is not necessary to construct a condition table.

The Equational All-Definitions Criterion

This criterion is trivial. It is not necessary to select any condition for this criterion. It is even not necessary to transform complex conditional equations into simple conditional equations in the pre-test compilation. Any single execution with input values for `x1` and `x2` will satisfy the criterion.

5.6.3 Selecting Test Input Values (step(3))

The tester finds test input values that satisfy a single or conjunctions of the conditions. These are the conditions which were selected in step (2). Each condition in the original array graph (of which edges are not deleted) may depend on input variables, interim variables and subscript values. The tester may backtrack along causality paths from the selected condition node to the respective input variable nodes. Finding input variable values and subscript values that satisfy the condition may be complex. This is an oracle operation that is to be done by the human tester. This operation is illustrated below for the respective criterion.

The Equational All-Paths Criterion

As shown in Figure 5.3, the tester selects first one condition at a time and finds test input values that satisfy the selected condition. The test is then applied. The order of selecting single conditions may reduce the number of tests. For example, if the tester finds test input values for satisfying `cond3(i)`, then it is guaranteed from the condition table (Table 5.4) that at least three rows will be marked by the single test execution. Namely, the test execution will satisfy row 3, row 5 or row 6, and

Single Condition	Min. no. of rows marked
cond1	1
cond2	1
cond3(i)	3
cond4(i)	3

Table 5.6: Single conditions and their minimum number of marking rows of the condition table.

Test No.	x1	x2
1	32	14
2	14	32

Table 5.7: Test input values.

row 1 or row 2 of the condition table. This is because **cond1** or **cond2** must be in the paths from **x1** and **x2** to **cond3(i)**. Thus an examination of the condition table may provide information about reducing the number of tests by ordering the conditions to be satisfied. Table 5.6 shows the minimum number of rows of Table 5.4 that will be marked by satisfying each single condition. Thus, generally, it is advantageous to select conditions that are member of conjunctions of multiple conditions.

The tester first finds test input values to satisfy **cond3(i)**. The tester must backtrack from **cond3(i)** along the paths in the array graph to **x1** and **x2**. Suppose `i = 4`. The subscript value needs then to be retraced from `i = 4` to `i = 1`. The tester finds a causality path from **cond3(4)** to **x1** and **x2** for the backtracking. The path is shown in Figure 5.12. The conditions, **cond1**, **cond3(2)**, **cond4(3)** and **cond3(4)**, are defined as follows:

(a) `cond1 = (x1 > x2)`

(b) `cond3(2) = (x1 > x2)`

(c) `cond4(3) = ^((x1 - x2) > x2)`

(d) `cond3(4) = (x1 - x2) > (x2 - (x1 - x2))`

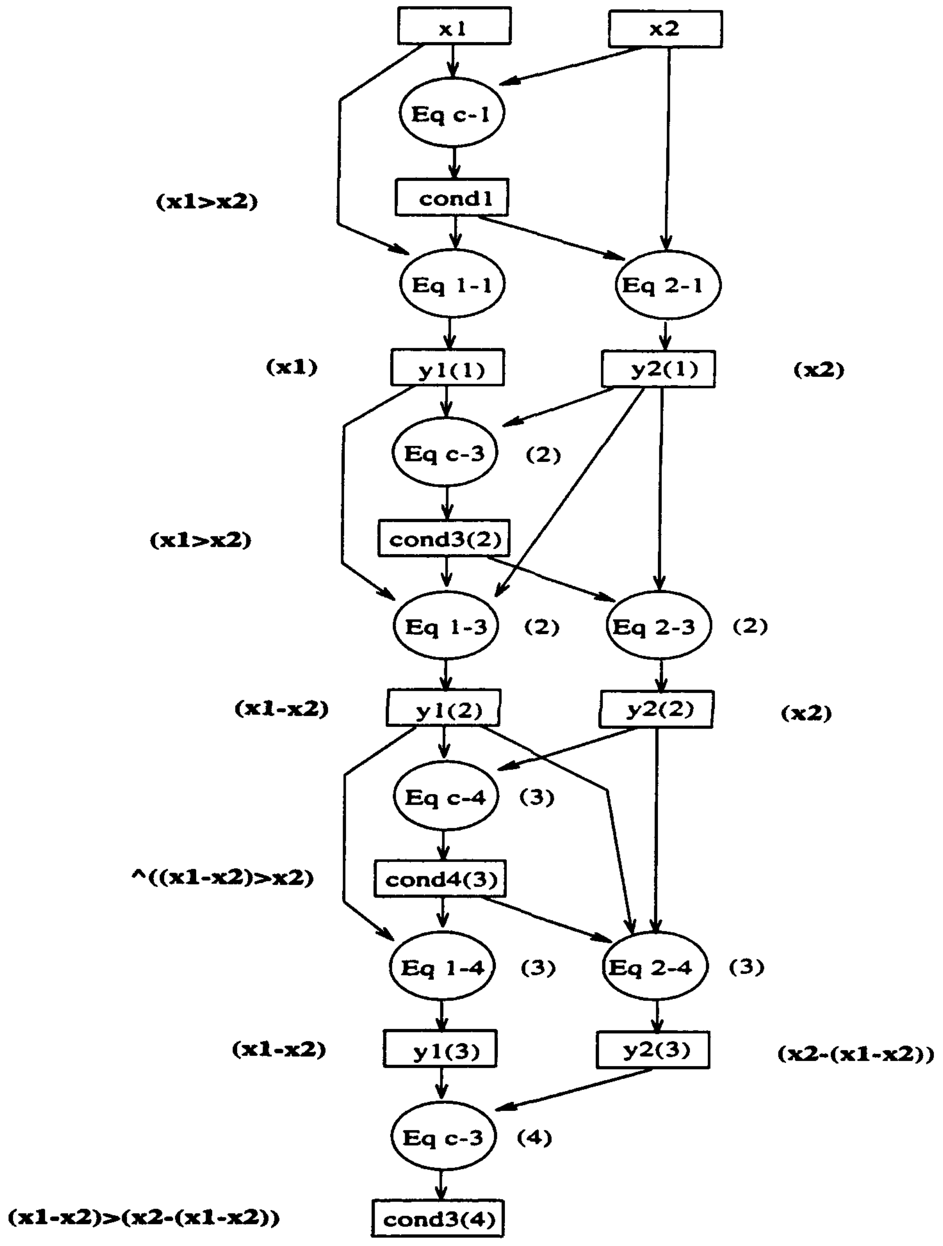


Figure 5.12: A causality path.

As reviewing the causality path in Figure 5.12, the tester can conclude that the conditions must be satisfied conjunctively. Namely,

```
cond1 & cond3(2) & cond4(3) & cond3(4).
```

Thus the following conditional expression must be satisfied to satisfy `cond3(4)`:

```
(x1 > x2) & ~((x1 - x2) > x2) & ((x1 - x2) > (x2 - (x1 - x2))).
```

The expression can be reduced as follows:

```
(3/2) * x2 < x1 <= 2 * x2.
```

The tester may select test input values `x1 = 25` and `x2 = 14`. By symmetry, `cond4(3)` is satisfied if

```
(3/2) * x1 < x2 <= 2 * x1.
```

Test input values of `x1 = 14` and `x2 = 25` can satisfy `cond4(i)` for `i = 4`. Two tests with the test input values are listed in Table 5.7.

The Equational All-Du-Paths Criterion

The tester examines the condition table in Table 5.5. The test input values in Table 5.7 can satisfy for this criterion.

The Equational All-Uses Criterion

The tester must find test input values for traversing the four single conditions. This criterion will be satisfied by test executions with the test input values in Table 5.7.

The Equational All-Definitions Criterion

Any single execution will satisfy this criterion. Either Test 1 or Test 2 of Table 5.7 completes the testing for this criterion.

Test No.	i	x1	x2	y1(i)	y2(i)	END.y1(i)	z
1	1	25	14	25	14	FALSE	
	2	25	14	11	14	FALSE	
	3	25	14	11	3	FALSE	
	4	25	14	8	3	FALSE	
	5	25	14	5	3	FALSE	
	6	25	14	2	3	FALSE	
	7	25	14	2	1	FALSE	
	8	25	14	1	1	TRUE	1
2	1	14	25	14	25	FALSE	
	2	14	25	14	11	FALSE	
	3	14	25	3	11	FALSE	
	4	14	25	3	8	FALSE	
	5	14	25	3	5	FALSE	
	6	14	25	3	2	FALSE	
	7	14	25	1	2	FALSE	
	8	14	25	1	1	TRUE	1

Table 5.8: Test results from Test 1 and Test 2.

5.6.4 Execution (step (4))

The tester enters the selected input values for the testing in a source file. The input values in the file must be consistent with their data types and formats as defined in the specification. A graphical user interface facilitates the process of entering and formatting input values.

The equational specification is then executed with the selected test input values. The results are shown in Table 5.8. After Test 1 and Test 2, all of the condition nodes are shaded and all rows of the condition table are marked. Thus the equational all-uses criterion is satisfied. The equational all-du-paths and the equational all-paths criteria are also satisfied.

5.6.5 Evaluation of Results (step (5))

The results from the test execution must be evaluated by the tester to find if there are errors in the equational specification. This is step (5) of Figure 5.3. It is assumed that the tester can determine whether the results are correct or not with respect to the given input values. This is the oracle assumption.

The tester should evaluate the test results preferably by using a method which is independent of that used in the specification. The tester may "reverse" the algorithm in the specification. For example, the tester may evaluate the results as follows. Suppose the output of the test execution with `x1` and `x2` is `z`. Let `z1 = x1 / z` and `z2 = x2 / z`. If `z > z1` or `z > z2`, then the result is correct, namely `z = GCD(x1, x2)`. Otherwise, the tester picks the number of `z1` and `z2` and finds if they have a common factor greater than `z`.

The evaluation process is not simple when either `x1` or `x2` are very large. Therefore it is easier to have small numbers to test the GCD specification.

5.7 Conclusion

The objective of software testing is to find if there are errors in a program (procedural. equational, etc.), with respect to its requirements. The program is executed with test input values and the respective results are evaluated. The testing focuses on analysis of graphs that represent the program. An array graph can be used for the Equational Visual testing and a control flow graph can be used for the procedural Testing.

Exhaustive testing, which tries all possible combination of all input values for the testing, is impractical. Therefore, a notion of adequacy criteria was introduced with less testing requirements. The adequacy criteria are defined in terms of paths of the underlying graphs, e.g., the array graph or the control flow graph. The criteria require selection of test input values that force traversal of selected paths in the graphs. The adequacy criteria can be classified in terms of their requirements of path traversals. These also define the order of their strictness. The selection of test input values is based on satisfying conditions in the programs.

The Equational Visual Testing and the Procedural Testing can be compared in the following aspects:

(1) **Adequacy Criteria**:

The test adequacy criteria and their testing processes defined for the Equational Visual Testing may be applied more generally than the Procedural Testing. The equational all-paths criterion defines a finite number of source-to-target paths based on acyclic array graphs (the equational compiler automatically opens an MSCC of the array graph). This makes the criterion practical, while the procedural all-paths criterion requires traversal of undefined number of paths.

The equational all-du-paths criterion defines at most two-edge paths, from an equation node (where a variable is defined) to its corresponding equation nodes (where the variable is used), which can be traversed by satisfying at most two conditions. This reduces labor of the testing in length of conjunction of the conditions for which test input values must be found. Note that a du-path may consist of multiple edges in the procedural testing.

The equational all-uses criterion is based on a single-edge path from a variable node to an equation node (where the variable is used). The path can be traversed by satisfying at most one condition. This reduces labor of the testing in length of conjunction of the conditions.

Satisfying the equational all-definitions criterion is trivial. Any single execution will traverse all definition paths. On the other hand, the procedural all-definitions criterion requires to traverse multiple assignment statements which define and update variables.

(2) **Visualization**:

The Equational Visual Testing is based on an array graph which directly represents data flow. This facilitates program understanding which is essential to program testing. As discussed, the array graph has constraints which make it simple to define the adequacy criteria and perform the testing process.

The condition nodes of the array graph are shaded when they are satisfied during test execution. The conjunctions of the conditions are listed in the condition table. The rows of the table are marked after test execution, according to the test results. These visualize progress of the testing procedure.

The human tester (a) selects test input values for satisfying specific conditions, (b) determines if a path is feasible, (c) evaluates test results with respect to the selected test input values and the test requirements. These are oracle assumptions. The visualization facilitates performing the orcle operations. All paths in the array graph visualizes causality paths. Thus the paths for backtracking from a specific condition to the respective input variables are explicitly visualized in the array graph. The tester can retrace the paths from the condition to the input variable nodes as selecting test input values. Each path can be traversed only if the conjunction of all conditions on the path is satisfied. If any of the condition is unsatisfiable, the path is not traversable. The conditions are represented by condition nodes of the array graph. The human tester can also refer to a specific iteration. This is possible because every elements of array variables are defined and the interim values are recorded during test execution. These reduce the labor of selecting test input values and evaluating test results.

The visualization technique can be applied to the Procedural Testing. The the progress of testing and code coverage can be visualized.

Chapter 6

Program Verification

6.1 Introduction

A MODEL specification is based on regular and boolean algebras that can be learned from high school. Formal verification of the correctness of a MODEL specification utilizes only algebraic manipulation of equations, namely **equational reasoning**. It is based on deduction rules by which equations can be deduced from the MODEL equations and general algebraic laws. A MODEL calculus is defined for the program verification.

This chapter is organized as follows: The MODEL calculus uses a fragment of MODEL. It uses general algebraic laws about arithmetics and logical equivalence and deduction rules which manipulate equations during the verification. They are presented in Section 6.2. An example is given in Section 6.3 of verifying the correctness of a MODEL specification under this calculus.

6.2 MODEL Calculus

The MODEL calculus is an equational calculus, that is, a calculus whose formulae are equations. An equation is an equality between two expressions. An expression is an algebraic term built over *signature*, Ω, (a set of operation symbols) which is presented in Section 6.2.1. The equations occurring in MODEL specifications are part of this calculus. The calculus contains another collection of equations which are not dependent on specific MODEL specifications. They are called general algebraic laws and formalized in Section 6.2.2. Given a MODEL specification, S, the MODEL calculus has deduction rules for deducing equations that follow from the equations of S and the general laws. For the deduced equation, e, a derivation tree is formulated under the deduction rules; the label of the root node is e; the label of each node matches the conclusion of some rule whose premises match the children of the node; the labels of the leaf nodes can be equations of S or the general laws. The derivation tree is a proof of the equation, e in equational reasoning. Section 6.2.3 presents the deduction rules.

6.2.1 Basic Notions

The syntax of the MODEL calculus is quite similar to that of the MODEL language. The syntactic similarity enables equations in a MODEL specification to be a part of the calculus. Note that the MODEL calculus allows function symbols in LHS of an equality, while the MODEL language does not. Such extension increases expressive power of the calculus.

An equation an equality between a left-hand side expression a right-hand side expression. A semicolon is used as a delimiter between equations:

$$Equation ::= LHSExpr = RHSExpr\ ;$$

An LHS expression can be a scalar variable, an array variable, or a function symbol:

$$LHSExpr ::= ScalVar \mid ArrayVar(SubExpr) \mid UsrFunSym(ArgList)$$

$$FunSym ::= UsrFunSym \mid BltnFunSym$$

$$SubExpr ::= SubElem[, SubExpr]$$

$$ArgList ::= ScalVar[, ArgList] \mid ArrayVar(SubExpr)[, ArgList]$$

A scalar variable ($ScalVar$) denotes a single value. An array variable ($ArrayVar(SubExpr)$) is indexed by a subscript expression, $SubExpr$. A subscript expression ($SubExpr$) is a list of subscript elements ($SubElem$) which is a special case of an arithmetic expression, that is, an arithmetic expression on natural numbers, **N**. A user-defined function symbol ($UsrFunSym$) is a function symbol ($FunSym$) which can be either user-defined or built-in ($BltnFunSym$). A user can provide his own function with a function name ($UsrFunSym$), its arguments ($ArgList$) and its definition in terms of an RHS expression. The rank (or arity) of the function is equal to the number of arguments in its argument list. Some frequently used common functions are called built-in functions:

(1) **sum (array-var)**:

computes a sum of values of each elements of array variable **array-var**.

(2) **max** ($var_1, \ldots, var_n$):

finds the maximum values among $var_1, \ldots, var_n$.

(3) **min** ($var_1, \ldots, var_n$):

selects the minimum values among $var_1, \ldots, var_n$.

(4) **sort (array-var, order)**:

returns a 1-dimensional array containing sorted elements of a 1-dimensional array variable **array-var** in the specified order: ascending when **order** = 1 and descending when **order** = 0.

Their names are reserved and their ranks are fixed. Since they cannot be redefined by users, they cannot be an LHS expression.

An RHS expression can be an arithmetic expression, a boolean expression, or a conditional expression:

$RHSExpr ::= ArithExpr \mid BoolExpr \mid CondExpr$

$CondExpr ::=$ **IF** $BoolExpr$ **THEN** $RHSExpr$ [**ELSE** $RHSExpr$]

An arithmetic expression ($ArithExpr$) is constructed over signature, $\Omega_q \overset{\text{def}}{=} \{+, -, *, /, \mathtt{Q}\}$, where $\mathtt{Q}$ denotes rational numbers, and the *ranks* of the operation symbols are $r(+) = 2$, $r(*) = 2$, $r(-) = 2$, $r(/) = 2$ and $\forall q \in \mathtt{Q}$, $r(q) = 0$. A boolean expression ($BoolExpr$) is built over signature, $\Omega_b \overset{\text{def}}{=} \{<, \leq, >, \geq, =, \neq, \wedge$ (and), $\vee$ (or), $\neg$ (not), $TRUE$, $FALSE$ $\}$, where $r(<) = 2$, $r(\leq) = 2$, $r(>) = 2$, $r(\geq) = 2$, $r(=) = 2$, $r(\neq) = 2$, $r(\wedge) = 2$, $r(\vee) = 2$, $r(\neg) = 1$, $r(TRUE) = 0$ and $r(FALSE) = 0$.

It follows that the signature, Ω, includes those signatures, Ω_q and Ω_b, **IF-THEN**, **IF-THEN-ELSE** and the function symbols. The RHS expressions are built over the signature with the scalar and array variables ($ScalVar$ and $ArrayVar$) and the function symbols ($FunSym$).

6.2.2 General Algebraic Laws

The general algebraic laws (or axioms) of the calculus are expressed as equations. The laws of arithmetic operations such as addition (+), subtraction (−), multiplication (∗) and division (/) are specified. Logical equivalence laws also presented as equations.

Laws of Arithmetic Operators

(1) **Commutativity:**

$$(x + y) = (y + x)$$

$$(x * y) = (y * x)$$

(2) **Associativity:**

$$x + (y + z) = (x + y) + z$$

$$x * (y * z) = (x * y) * z$$

(3) **Distributivity:**

$$x * (y + z) = (x * y) + (y * z)$$

Laws of Logical Operators

(1) **Commutativity:**

$$(x \wedge y) = (y \wedge x)$$

$$(x \vee y) = (y \vee x)$$

$$(x = y) = (y = x)$$

(2) **Associativity:**

$$(x \wedge (y \wedge z)) = ((x \wedge y) \wedge z)$$

$$(x \vee (y \vee z)) = ((x \vee y) \vee z)$$

(3) **Distributivity:**

$$(x \vee (y \wedge z)) = ((x \vee y) \wedge (y \vee z))$$

$$(x \wedge (y \vee z)) = ((x \wedge y) \vee (x \wedge z))$$

(4) **De Morgan's Laws:**

$$\neg(x \wedge y) = (\neg x \vee \neg y)$$

$$\neg(x \vee y) = (\neg x \wedge \neg y)$$

(5) **Negation:** $\neg(\neg x) = x$

(6) **Excluded Middle:** $(x \vee \neg x) = TRUE$

(7) **Contradiction:** $(x \wedge \neg x) = FALSE$

(8) **Implication:**

$$(\textbf{IF } TRUE \textbf{ THEN } x) = x$$

$$(\textbf{IF } TRUE \textbf{ THEN } x \textbf{ ELSE } y) = x$$

$$(\textbf{IF } FALSE \textbf{ THEN } x \textbf{ ELSE } y) = y$$

(9) **OR-simplification:**

$$(x \vee x) = x$$

$$(x \vee TRUE) = TRUE$$

$$(x \vee FALSE) = x$$

$$(x \vee (x \wedge y)) = x$$

(10) **AND-simplification:**

$$(x \wedge x) = x$$

$$(x \wedge TRUE) = x$$

$$(x \wedge FALSE) = FALSE$$

$$(x \wedge (x \vee y)) = x$$

6.2.3 Deduction

With a set of equations of a MODEL specification and the equations of the general laws, equational reasoning is performed to deduce equations in the proof goals which usually concern the correctness of the MODEL specification. For each equation of the proof goals, a derivation tree is formulated as one of deduction rules is applied to each step of deduction. Leaf nodes of the tree are equations of the MODEL specification or the general algebraic laws. The root node of the tree is the deduced equation which is one of the proof goals. The completed derivation tree denotes a proof of the deduced equation. We introduce a set of basic deduction rules of equational

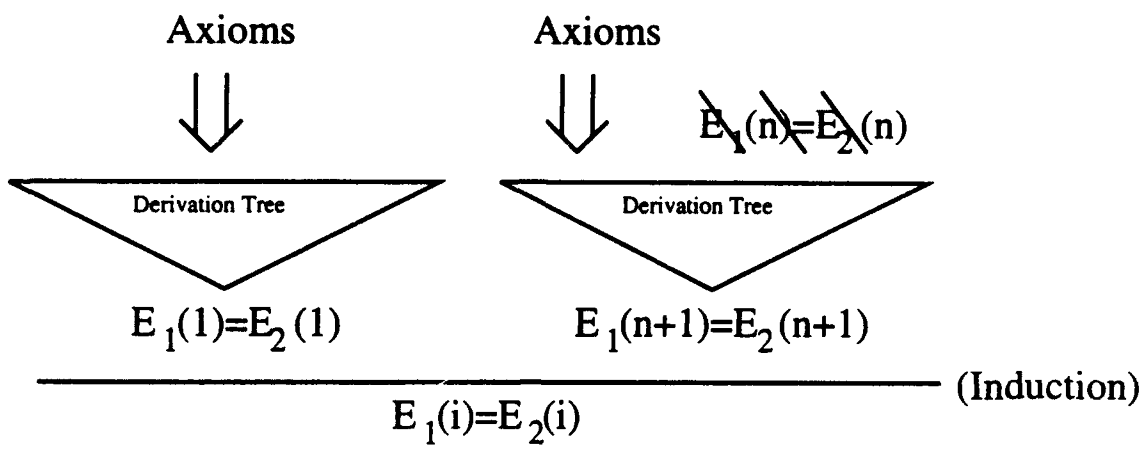


Figure 6.1: Induction

reasoning, induction, a method of case analysis and tactics in this section. The implementation of these techniques are discussed in this chapter.

Basic Deduction Rules

Each branch of derivation trees can be built from the following basic deduction rules of equational reasoning, where E_1, E_2 and E_3 are expressions:

(1) **Reflexivity:**

$$\frac{}{E_1 = E_1}$$

(2) **Symmetry:**

$$\frac{E_1 = E_2}{E_2 = E_1}$$

(3) **Transitivity:**

$$\frac{E_1 = E_2 \quad E_2 = E_3}{E_1 = E_3}$$

(4) **Replacement:**

$$\frac{E_1 = E_2}{E_3[E_1/x] = E_3[E_2/x]}$$

(5) **Substitutivity:**

$$\frac{E_1 = E_2}{E_1[E_3/x] = E_2[E_3/x]}$$

Induction

A proof technique, *induction*, is discussed. In general, a property $P(x)$ is said to be proven, that is, $\forall x, P(x)$ holds, "by induction" if both $P(0)$ (called *basis*) and $\forall n \in \mathtt{N}, P(n) \Rightarrow P(n+1)$ (called *step*) are proven to be true.

A property of a MODEL specification, S, can be expressed as an equation, $E_1(i) = E_2(i)$, which can be indexed by a subscript, i. The equation is interpreted as follows: for all values of the subscript i, the two expressions, $E_1(i)$ and $E_2(i)$, have the same values. The equation can be proven by examining the basis case $(i = 1)$ and the step case $(\forall i > 1)$. The proof can be formulated by constructing two derivation trees, as shown in Figure 6.1. If the trees for the two cases are successfully constructed through equational reasoning, it is concluded that $\forall i \geq 1$, $E_1(i) = E_2(i)$ is derivable, that is, there exists a derivation tree for the equation using equations of S and the general laws.

Induction in the MODEL calculus is defined as follows:

Definition: An equation $E_1(i) = E_2(i)$ is derivable, if the followings are derivable:

(basis) $E_1(1) = E_2(1)$.

(step) Given $E_1(n) = E_2(n)$ as an additional axiom, $E_1(n+1) = E_2(n+1)$.

Note that three lines of *cross-out* on the equation $E_1(n) = E_2(n)$ in Figure 6.1. They denote the fact that the equation is a temporary assumption provided to derive $E_1(n+1) = E_2(n+1)$. That is, the equation $E_1(n) = E_2(n)$ will be discarded after induction. Also, the equation $E_1(n+1) = E_2(n+1)$ should be discarded after induction.

Case Analysis

Another derivation rule is *case analysis*. Suppose an equation contains a boolean expression B consisting of variables and constants. Since such a boolean expression

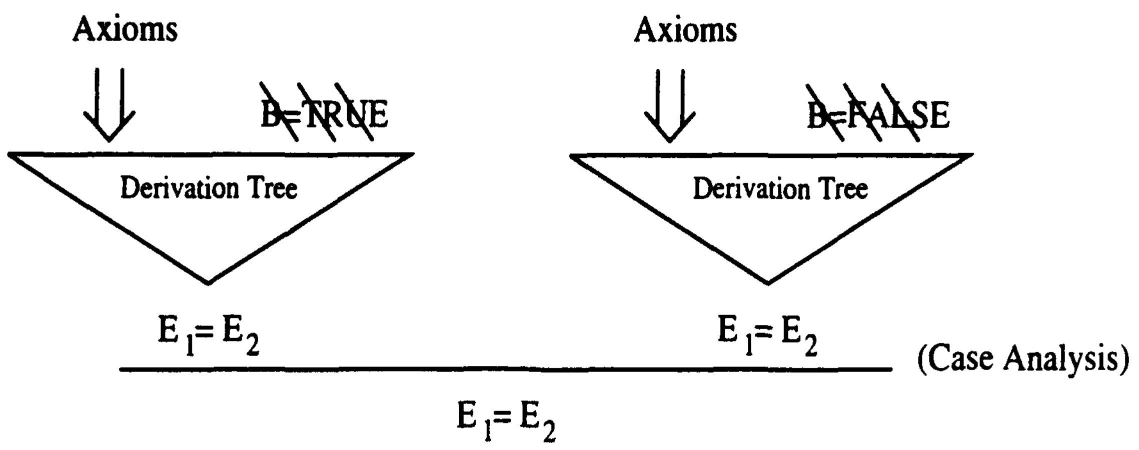


Figure 6.2: Case Analysis

cannot be evaluated unless all the values of the variables are known, the equation cannot be evaluated either. However, the equation may be simplified and hopefully evaluated if the truth value of the boolean expression is recognized. Thus we try to simplify the equation separately for two possible cases of the truth value of the boolean expression, namely $\mathtt{B} = TRUE$ and $\mathtt{B} = FALSE$. If the two results are same, say $\mathtt{E}_1 = \mathtt{E}_2$, we can conclude that the result is always derivable regardless of the values of B. That is, the following logical statement is true:

$$(((\mathtt{B} = TRUE) \Rightarrow (\mathtt{E}_1 = \mathtt{E}_2)) \wedge ((\mathtt{B} = FALSE) \Rightarrow (\mathtt{E}_1 = \mathtt{E}_2))) \Rightarrow (\mathtt{E}_1 = \mathtt{E}_2)$$

As shown in Figure 6.2, the idea is implemented as follows:

(1) Supply temporary axioms $\mathtt{B} = TRUE$ and $\mathtt{B} = FALSE$.

(2) Derive $\mathtt{E}_1 = \mathtt{E}_2$ for each case.

(3) If the derivation is successful, that is, the roots of the trees are identical, then conclude that $\mathtt{E}_1 = \mathtt{E}_2$ is derivable.

(4) Delete the temporary axioms on which the cross-out lines are marked.

Tactic

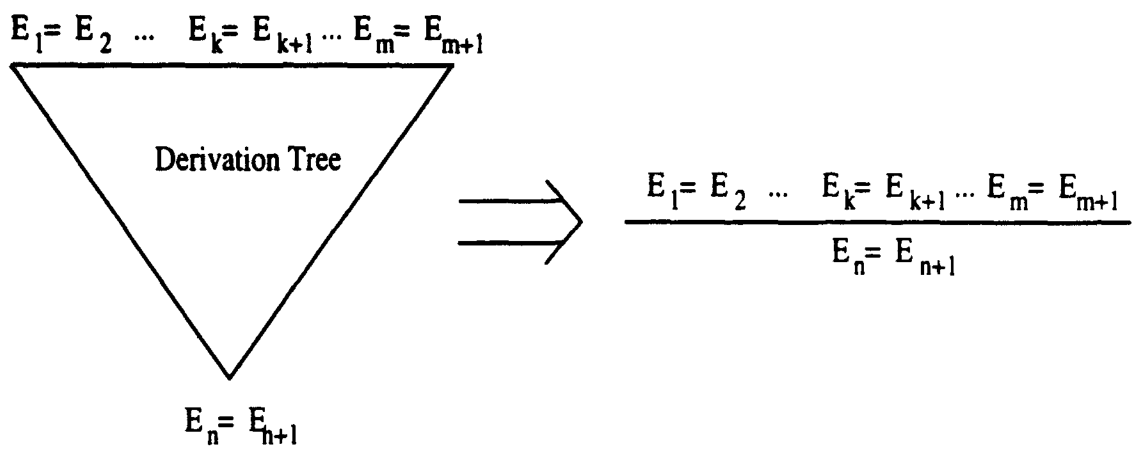

$E_1 = E_2 \; ... \; E_k = E_{k+1} \; ... \; E_m = E_{m+1}$
Derivation Tree
$E_n = E_{n+1}$
$E_1 = E_2 \; ... \; E_k = E_{k+1} \; ... \; E_m = E_{m+1}$
$E_n = E_{n+1}$

Figure 6.3: A derivation tree is reduced as a single rule called a tactic.

To reduce a size of a proof (or a derivation tree), a series of deduction rules, which contains multiple deduction steps and is used many times during the whole verification process, can be merged into a new rule called a *tactic.* Since a number of rule applications can be replaced by a single tactic application, the size of the whole proof can be reduced. A tactic can be envisaged as a macro in assembly language programming.

A tactic is virtually another deduction rule derived from either a series of basic deduction rules or a series of basic deduction rules and other tactics. Once a tactic is defined, it is regarded as a new deduction rule. Therefore a tactic should be "correctly" derived from other deduction rules. Its correctness can be proven by building a derivation tree for the tactic through equational reasoning.

Suppose a derivation tree is constructed as shown in Figure 6.3, that is, all of its leaf node equations (or premises) are $E_1 = E_2$, ..., $E_k = E_{k+1}$, ..., $E_m = E_{m+1}$, and its root node equation (or conclusion) is $E_n = E_{n+1}$. We can formulate a single rule by merging the deduction rules as illustrated in Figure 6.3 such that its premises are all of the leaf nodes, $E_1 = E_2$, ..., $E_k = E_{k+1}$, ..., $E_m = E_{m+1}$ and its conclusion is the root node, $E_n = E_{n+1}$.

Consider the following derivation tree:

$$\dfrac{\dfrac{a(i) = b(i) \quad b(i) = c(i) + d(i)}{a(i) = c(i) + d(i)}\ (Trans)}{a(i+1) = c(i+1) + d(i+1)}\ (Subs, [(i+1)/i])$$

We can generalize the derivation. That is, we can merge the rule of **Transitivity**:

$$\frac{E_1 = E_2 \quad E_2 = E_4}{E_1 = E_4}$$

and **Substitutivity**:

$$\frac{E_1 = E_4}{E_1[E_3/x] = E_4[E_3/x]}$$

to bear a new tactic as follows:

$$\frac{E_1 = E_2 \quad E_2 = E_4}{E_1[E_3/x] = E_4[E_3/x]}$$

6.3 Example

This section demonstrates the methodology of verification based on equational reasoning. Given a set of equations of a MODEL specification, proof goals, which are also equations formulated by users to verify the correctness of the specification, are to be proved as the deduction rules are applied. A proof of each equation is a derivation tree formulated during equational reasoning.

Most verification methodologies keep track of changes of execution states that are values of program variables [Man74]. Since a variable has a single value in equational languages such as MODEL, we never trace changes of execution states (values of program variables) during the verification of an equational language program.

Unlike other equational languages such as Lucid [AW76, AW77], no temporal operators (**first, next, as soon as** etc.) are necessary in MODEL. In its calculus, passage of time [MP81, OL82, Lam83, Kro87] is replaced by the notion of implicit universal and existential quantifiers (the conditions of existence of variables) and "firing" of equations: The order of subscripts corresponds to the ordering of firing equations to determine variables. In other cases, "firing" of equations may be in parallel.

Usually, programs being tested are written in a programming language that a user is familiar with. Their axioms and proof goals are normally expressed in the *object language*[1] of the formal verification system is often very formal and rigid. It has been pointed out that the user must spend much time and energy translating the "natural" descriptions of the programs, the requirements and the proof goals into the "complicated" logical forms of the object language [Lin88]. Ideally, the object language of the formal verification system should be close to the language of programs. In this methodology, the object language is exactly same as the specification language.

The problem of finding a greatest common divisor (GCD) of two positive integers in Chapter 2 is chosen as an illustrating example. Proving the correctness of the specification requires three inputs: the specification equations, axioms and proof goals. These are discussed in the following. Next, the formal proof of the correctness is presented.

The user enters axioms about the specification. They define the required behaviors of the specification and they become axioms in the verification system. In this particular example, the following axiom about a gcd of two positive integers v and w is the basis of the Euclid algorithm:

```
Axiom 1: gcd(v,w) =  IF v = w THEN v
                     ELSE IF v < w THEN gcd(v,w-v)
                                   ELSE gcd(v-w,w)
```

Next, goals for program verification are specified as proof goals. The proof goals consist of goals and their subgoals. The user is responsible for decomposing each proof goal into a sequence of subgoals. He must assure that a derivation tree (proof) of each goal is formulated by combining derivation trees (proofs) of its subgoals. The derivation trees of subgoals are constructed through equational reasoning. The

[1]a logical language in which propositions are expressed and reasoned about

derivation trees are, in fact, proofs of their corresponding subgoals. The following proof goal is prepared for this example:

Goal I. $z = gcd(x1, x2)$

Subgoal 1. $gcd(x1, x2) = gcd(y1(1), y2(1))$

Subgoal 2. $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$

Subgoal 3. $z = gcd(y1(SIZE.y1), y2(SIZE.y1))$

The verification process aims to prove that the specification correctly computes the gcd, z, of the input variables, $x1$ and $x2$. The first subgoal is to prove that the gcd of the inputs is equal to the gcd of the first elements of temporal variables, $y1(1)$ and $y2(1)$. The second subgoal is to verify the correct computation between any consecutive elements of $y1(i)$ and $y2(i)$. The third one asserts that the output, z, is correctly computed. The proof goal, Goal I, is then proven by combining the derivation trees (or proofs) of the subgoals.

The deduction rules are applied during the process of verification. The subgoals presented in the proof goals are proven one by one. The followings are the steps of the verification. For each step of verification, the names of the deduction rules used for the verification step are given:

```
Subgoal 1. gcd(x1,x2) = gcd(y1(1),y2(1)):
a. (from Eq 1)
   y1(i)=IF i=1 THEN IF x1>x2 THEN x1 ELSE x2
                ELSE IF y1(i-1)>y2(i-1) THEN y1(i-1)-y2(i-1)
                                         ELSE y1(i-1)
   ----------------------------------------------------(Subs,[1/i])
   y1(1)=IF 1=1 THEN IF x1>x2 THEN x1 ELSE x2
                ELSE IF y1(1-1)>y2(1-1) THEN y1(1-1)-y2(1-1)
                                         ELSE y1(1-1)
   ----------------------------------------------------(Subs,[TRUE/(1=1)])
   y1(1)=IF TRUE THEN IF x1>x2 THEN x1 ELSE x2
```

```
                    ELSE IF y1(1-1)>y2(1-1) THEN y1(1-1)-y2(1-1)
                                                 ELSE y1(1-1)
b. (from  (Implication) in the general axioms)
   IF TRUE THEN x ELSE y = x
   -------------------------(Subs,[(IF x1>x2 THEN x1 ELSE x2)/x])
   IF TRUE THEN IF x1>x2 THEN x1 ELSE x2 ELSE y
   = IF x1>x2 THEN x1 ELSE x2
   -------------------------(Subs,[(IF y1(1-1)>y2(1-1) THEN y1(1-1)-y2(1-1)
                                                   ELSE y1(1-1))/y])
   IF TRUE THEN IF x1>x2 THEN x1 ELSE x2
           ELSE IF y1(1-1)>y2(1-1) THEN y1(1-1)-y2(1-1)
                                   ELSE y1(1-1)
   = IF x1>x2 THEN x1 ELSE x2
c.       (a)           (b)
   ------------------------------(Trans)
   y1(1)=IF x1>x2 THEN x1 ELSE x2
```

Note that the general law of **Implication** says: (**IF** $TRUE$ **THEN** x **ELSE** y) $= x$. Suppose we have equations, $E_1 =$ **IF** $TRUE$ **THEN** E_2 **ELSE** E_3. The RHS expression **IF** $TRUE$ **THEN** E_2 **ELSE** E_3 is equal to E_2 by the general law of **Implication**. Thus the following derivation is possible:

```
        (Given)                  (from General Law of Implication)
  E1 = IF TRUE THEN E2 ELSE E3   (IF TRUE THEN E3 ELSE E4) = E3
  ------------------------------------------------------------(Trans)
                           E1 = E3
```

The derivation can be simplified as a tactic called **TRUE-Eval** by removing the general law of **Implication**:

TRUE-Eval:

$$\frac{E_1 = \text{IF } TRUE \text{ THEN } E_2 \text{ ELSE } E_3}{E_1 = E_3}$$

Next, an RHS expression for `y2(1)` is derived:

```
d. (from Eq 2)
   y2(i)=IF i=1 THEN IF x1>x2 THEN x2 ELSE x1
                ELSE IF y1(i-1)>y2(i-1) THEN y2(i-1)
                                        ELSE y2(i-1)-y1(i-1)
   ------------------------------------------------------(Subs,[1/i])
   y2(1)=IF 1=1 THEN IF x1>x2 THEN x2 ELSE x1
                ELSE IF y1(1-1)>y2(1-1) THEN y1(1-1)
                                        ELSE y2(1-1)-y1(1-1)
   ------------------------------------------------------(Subs,[TRUE/(1=1)])
   y2(1)=IF TRUE THEN IF x1>x2 THEN x2 ELSE x1
                ELSE IF y1(1-1)>y2(1-1) THEN y2(1-1)
                                        ELSE y2(1-1)-y1(1-1)
   ------------------------------------------------------(TRUE-Eval)
                y2(1)=IF x1>x2 THEN x2 ELSE x1
```

To simplify the two deduced equations, (c) and (d), that is, to evaluate the boolean expression, **x1>x2**, it is required for a user to provide a way to evaluate it. In this particular example, he provides the following equations which are assured to be able to cover the possible values of the boolean expression and mutually exclusive: x1>x2 = TRUE and x1>x2 = FALSE. For each equation, a derivation tree can be formulated. If all the trees have same conclusions, it is interpreted as if the conclusions are derived from the equations, (c) and (d). First, it is assumed that **x1** is greater than **x2**, that is, **x1>x2=TRUE**:

```
CASE x1>x2:
e.               (c)
   --------------------------------(Subs,[TRUE/(x1>x2)])
   y1(1) = IF TRUE THEN x1 ELSE x2
   --------------------------------(TRUE-Eval)
           y1(1) = x1
           ----------(Symm)
           x1 = y1(1)
```

```
    --------------------------(Repl)
    gcd(x1,x2) = gcd(y1(1),x2)
f.                (d)
    -------------------------------(Subs,[TRUE/(x1>x2)])
    y2(1) = IF TRUE THEN x2 ELSE x1
    -------------------------------(TRUE-Eval)
            y2(1) = x2
            ----------(Symm)
            x2 = y2(1)
    -----------------------------(Repl)
    gcd(y1(1),x2) = gcd(y1(1),y2(1))
g.          (e)         (f)
    -----------------------------(Trans)
    gcd(x1,x2) = gcd(y1(1),y2(1))
```

Next, another case where **x1** is not greater than **x2** should be taken care of, that is, **x1>x2=FALSE**:

```
CASE x1<=x2:
h.                (c)
    -------------------------------(Subs,[FALSE/(x1>x2)])
    y1(1) = IF FALSE THEN x1 ELSE x2
```

A user may need a new tactic, **FALSE-Eval**, which is dual to **TRUE-Eval** and can be derived from the general axioms (**Implication**) using the deduction rules as follows:

FALSE-Eval:

$$\frac{E_1 = \text{IF } FALSE \text{ THEN } E_2 \text{ ELSE } E_3}{E_1 = E_3}$$

The tactic is applied to (h):

```
i.                (h)
            ----------(FALSE-Eval)
```

```
          y1(1) = x2
          ----------(Symm)
          x2 = y1(1)
   --------------------------(Repl)
   gcd(x1,x2) = gcd(x1,y1(1))
j.               (d)
   --------------------------------(Subs,[FALSE/(x1>x2)])
   y2(1) = IF FALSE THEN x2 ELSE x1
   --------------------------------(FALSE-Eval)
            y2(1) = x1
            ----------(Symm)
            x1 = y2(1)
   --------------------------------(Repl)
   gcd(x1,y1(1)) = gcd(y2(1),y1(1))
k.         (i)         (j)
   -----------------------------(Trans)
   gcd(x1,x2) = gcd(y2(1),y1(1))
```

Now, we can derive a new lemma, whose proof is omitted in this document, from `Axiom 1`, as follows:

`Lemma 1: gcd(x1,x2) = gcd(x2,x1)`

It follows that the following derivation is possible:

```
l. (by Lemma 1)
   gcd(y2(1),y1(1)) = gcd(y1(1),y2(1))
m.         (k)         (l)
   -----------------------------(Trans)
   gcd(x1,x2) = gcd(y1(1),y2(1))
```

It concludes that Subgoal 1, `gcd(x1,x2)=gcd(y1(1),y2(1))`, is proven.

Next, Subgoal 2, `gcd(y1(i),y2(i))=gcd(y1(i-1),y2(i-1))`, is considered. Note that `(i=1)=FALSE` for Subgoal 2. Thus the followings can be obtained:

Subgoal 2. `gcd(y1(i),y2(i))=gcd(y1(i-1),y2(i-1))`:

```
a. (from Eq 1)
   y1(i)=IF i=1 THEN IF x1>x2 THEN x1 ELSE x2
                ELSE IF y1(i-1)>y2(i-1) THEN y1(i-1)-y2(i-1)
                                        ELSE y1(i-1)
   --------------------------------------------------(Subs,[FALSE/(i=1)])
   y1(i)=IF FALSE THEN IF x1>x2 THEN x1 ELSE x2
                  ELSE IF y1(i-1)>y2(i-1) THEN y1(i-1)-y2(i-1)
                                          ELSE y1(i-1)
   --------------------------------------------------(FALSE-Eval)
   y1(i)=IF y1(i-1)>y2(i-1) THEN y1(i-1)-y2(i-1)
                            ELSE y1(i-1)
b. (from Eq 2: (Subs,[FALSE/(i=1)]) and (FALSE-Eval))
   y2(i)=IF y1(i-1)>y2(i-1) THEN y2(i-1)
                            ELSE y2(i-1)-y1(i-1)
```

Next, three cases, `y1(i-1)>y2(i-1)`, `y1(i-1)<y2(i-1)` and `y1(i-1)=y2(i-1)`, are considered. The first case, where `y1(i-1)>y2(i-1)=TRUE` holds, is:

```
CASE y1(i-1)>y2(i-1):
c.                    (a)
   ----------------------------------(Subs,[TRUE/(y1(i-1)>y2(i-1))])
   y1(i)=IF TRUE THEN y1(i-1)-y2(i-1)
                 ELSE y1(i-1)
   ----------------------------------(TRUE-Eval)
         y1(i)=y1(i-1)-y2(i-1)
         ---------------------(Symm)
         y1(i-1)-y2(i-1)=y1(i)
d. (from (b): (Subs,[TRUE/(y1(i-1)>y2(i-1))]), (TRUE-Eval) and (Symm))
         y2(i-1)=y2(i)
```

From `Assertion 1` and the equation, `y1(i-1)>y2(i-1)=TRUE`, which logically implies (y1(i-1)=y2(i-1))=FALSE, we can derive the following equation:

```
e. (from Axiom 1)
   gcd(v,w)=IF v=w THEN v
                   ELSE IF v>w THEN gcd(v-w,w)
                                ELSE gcd(v,w-v)
   -------------------------------------------(Subs,[y1(i-1)/v])
   gcd(y1(i-1),w)=IF y1(i-1)=w THEN y1(i-1)
                               ELSE
                  IF y1(i-1)>w THEN gcd(y1(i-1)-w,w)
                               ELSE gcd(y1(i-1),w-y1(i-1))
   -------------------------------------------(Subs,[y2(i-1)/w])
   gcd(y1(i-1),y2(i-1))=IF y1(i-1)=y2(i-1) THEN y1(i-1)
                                           ELSE
                        IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                           ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
   -------------------------------------------(Subs,[FALSE/(y1(i-1)=y2(i-1))])
   gcd(y1(i-1),y2(i-1))=IF FALSE THEN y1(i-1)
                                 ELSE
                        IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                           ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
   -------------------------------------------(FALSE-Eval)
   gcd(y1(i-1),y2(i-1))=IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                           ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
   -------------------------------------------(Subs,[TRUE/(y1(i-1)>y2(i-1))])
   gcd(y1(i-1),y2(i-1))=IF TRUE THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
   -------------------------------------------(TRUE-Eval)
   gcd(y1(i-1),y2(i-1))=gcd(y1(i-1)-y2(i-1),y2(i-1))
   -------------------------------------------(Subs,[y1(i-1)-y2(i-1)/y1(i)])
   gcd(y1(i-1),y2(i-1))=gcd(y1(i),y2(i-1))
```

```
f.                    (d)
   ----------------------------------(Repl)
   gcd(y1(i),y2(i-1))=gcd(y1(i),y2(i))
g.                  (e)     (f)
     ------------------------------------(Trans)
     gcd(y1(i-1),y2(i-1))=gcd(y1(i),y2(i))
     ------------------------------------(Symm)
     gcd(y1(i),y2(i))=gcd(y1(i-1),y2(i-1))
```

The second case is based on the equation, y1(i-1)>y2(i-1)=FALSE. Thus the following derivation is made:

```
CASE y1(i-1)<y2(i-1):
h.                    (a)
   ----------------------------------(Subs,[FALSE/(y1(i-1)>y2(i-1))])
   y1(i)=IF FALSE THEN y1(i-1)-y2(i-1)
                  ELSE y1(i-1)
   ----------------------------------(FALSE-Eval)
         y1(i)=y1(i-1)
         -------------(Symm)
         y1(i-1)=y1(i)
i. (from (b): (Subs,[FALSE/y1(i-1)>y2(i-1)]), (FALSE-Eval) and (Symm))

         y2(i-1)-y1(i-1)=y2(i)
```

Given Axiom 1, y1(i-1)>y2(i-1)=FALSE and y1(i-1)=y2(i-1)=FALSE, we can derive the following equation:

```
j. (from Axiom 1)
   gcd(v,w)=IF v=w THEN v
                ELSE IF v>w THEN gcd(v-w,w)
                             ELSE gcd(v,w-v)
   ------------------------------------------(Subs,[y1(i-1)/v])
```

```
gcd(y1(i-1),w)=IF y1(i-1)=w THEN y1(i-1)
                           ELSE
               IF y1(i-1)>w THEN gcd(y1(i-1)-w,w)
                            ELSE gcd(y1(i-1),w-y1(i-1))
-------------------------------------------(Subs,[y2(i-1)/w])
gcd(y1(i-1),y2(i-1))=IF y1(i-1)=y2(i-1) THEN y1(i-1)
                                        ELSE
                     IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                        ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
-------------------------------------------(Subs,[FALSE/(y1(i-1)=y2(i-1))])
gcd(y1(i-1),y2(i-1))=IF FALSE THEN y1(i-1)
                              ELSE
                     IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                        ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
-------------------------------------------(FALSE-Eval)
gcd(y1(i-1),y2(i-1))=IF y1(i-1)>y2(i-1) THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                                        ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
-------------------------------------------(Subs,[FALSE/(y1(i-1)>y2(i-1))])
gcd(y1(i-1),y2(i-1))=IF FALSE THEN gcd(y1(i-1)-y2(i-1),y2(i-1))
                              ELSE gcd(y1(i-1),y2(i-1)-y1(i-1))
-------------------------------------------(FALSE-Eval)
gcd(y1(i-1),y2(i-1))=gcd(y1(i-1),y2(i-1)-y1(i-1))
-------------------------------------------(Subs,[y2(i-1)-y1(i-1)/y2(i)])
gcd(y1(i-1),y2(i-1))=gcd(y1(i-1),y2(i))
k.                (h)
-----------------------------------(Repl)
gcd(y1(i-1),y2(i))=gcd(y1(i),y2(i))
l.              (j)     (h)
 -------------------------------------(Trans)
 gcd(y1(i-1),y2(i-1))=gcd(y1(i),y2(i))
 -------------------------------------(Symm)
```

```
gcd(y1(i),y2(i))=gcd(y1(i-1),y2(i-1))
```

The third case, `y1(i-1)=y2(i-1)`, turns out to be a condition of terminating the computation: Since the value of control variable, `END.y1(i)`, is defined as `TRUE` as soon as the condition, `y1(i)=y2(i)`, is satisfied, `Eq 4` is executed. It results in `z=y1(i)`, where `i=SIZE.y1` and the computation terminates. Thus `y1(i)` and `y2(i)` cannot be defined if `y1(i-1)=y2(i-1)`.

As discussed in Section 2.3.4, the rule of control variables in MODEL can be applied to the following lemma:

```
Lemma 2: END.y1(SIZE.y1)=TRUE
```

Next, a derivation tree for Subgoal 3 is constructed.

```
Subgoal 3. z=gcd(y1(SIZE.y1),y2(SIZE.y1)):
a. (from Eq 3)
           END.y1(i)=(y1(i)=y2(i))
   ----------------------------------------(Subs,[SIZE.y1/i])
   END.y1(SIZE.y1)=(y1(SIZE.y1)=y2(SIZE.y1))
   ----------------------------------------(Symm)
   (y1(SIZE.y1)=y2(SIZE.y1))=END.y1(SIZE.y1)
b. (by Lemma 2)
    END.x(SIZE.x)=TRUE
   --------------------(Subs,[y1/x])
   END.y1(SIZE.y1)=TRUE
c.          (a)     (b)
   ------------------------------(Trans)
   (y1(SIZE.y1)=y2(SIZE.y1))=TRUE
```

This conclusion of the reasoning results in the following lemma:

```
Lemma 3: y1(SIZE.y1)=y2(SIZE.y1)
```

A new equation is derived from `Eq 4` as follows:

```
d. (from Eq 4)
         z=IF END.y1(i) THEN y1(i)
   ------------------------------------(Subs,[SIZE.y1/i])
   z=IF END.y1(SIZE.y1) THEN y1(SIZE.y1)
   ------------------------------------(Subs,[TRUE/END.y1(SIZE.y1)])
        z=IF TRUE THEN y1(SIZE.y1)
        --------------------------(TRUE-Eval)
                z=y1(SIZE.y1)
```

It follows that:

```
e. z=y1(SIZE.y1)  y1(SIZE.y1)=y2(SIZE.y1)
   --------------------------------------(Trans)
               z=y2(SIZE.y1)
f.           (d)
   ---------------------------(Repl)
   gcd(z,z)=gcd(y1(SIZE.y1),z)
g.                    (e)
   --------------------------------------------(Repl)
   gcd(y1(SIZE.y1),z)=gcd(y1(SIZE.y1),y2(SIZE.y1))
h.             (f)      (g)
   -------------------------------------(Trans)
   gcd(z,z)=gcd(y1(SIZE.y1),y2(SIZE.y1))
```

The following lemma can be deduced form `Axiom 1`:

`Lemma 4: z=gcd(z,z)`

It follows that the following derivation is possible:

```
i. z=gcd(z,z) gcd(z,z)=gcd(y1(SIZE.y1),y2(SIZE.y1))
   ------------------------------------------------(Trans)
         z=gcd(y1(SIZE.y1),y2(SIZE.y1))
```

Thus Subgoal 3 is proven.

Chapter 7

Knowledge Acquisition

7.1 Introduction

This chapter describes knowledge acquisition for knowledge bases of a rule-based expert system via **rule extraction** from equations. It is based on tools of language translation, namely procedural-to-equational and equational-to-rule translators.

There is valuable expertise in existing programs such as algorithms and methods. It can be utilized through the following steps of rule extraction: (1) translation of existing programs to specifications written in an equational language and (2) translation of the specifications to expert system rules. It follows that the expertise in existing programs can be transferred to the expert system rules.

As illustrated in Figure 2.8, a procedural-to-equational language translator that produces MODEL specifications from existing procedural programs can be used in the automation of knowledge acquisition [PLK91]. The programs and the specifications are stored in the repository of the environment. Next, the equational-to-rule translator generates CLIPS rules from the MODEL specifications. The rules are accumulated in the knowledge base of a rule-based expert system.

The MODEL and the CLIPS languages are compared in Section 7.2. Section 7.3 explains the equational-to-rule translator. The translation process is illustrated based on translation of the GCD specification [1] into CLIPS rules in Section 7.4.

7.2 A MODEL Equation vs. a CLIPS Rule

How is an expert system rule similiar to a MODEL equation? As an example, consider the following equation, Eq 1 in Figure 2.1, Chapter 2:

```
Eq 1: y1(i) = IF i=1 THEN IF x1 > x2 THEN x1 ELSE x2
                          ELSE IF y1(i-1) > y2(i-1) THEN y1(i-1) - y2(i-1)
                                                     ELSE y1(i-1);
```

The equation accepts the input values of variables, $x1, x2, y1(i)$ and $y2(i)$, where $1 \leq i \leq SIZE.y1$ and computes the values of $y1(i)$ for $1 \leq i \leq SIZE.y1$. The existence condition of Eq 1 must be:

$$\forall i, 1 \leq i \leq SIZE.y1, (\exists x1, x2 | \exists y1(i-1), y2(i-1))$$

It means that the values of $x1$ and $x2$ OR $y1(i-1)$ and $y2(i-1)$ must be available when the value of $y1(i)$ is determined by the equation, Eq 1. The existence condition can be satisfied either $\exists x1, x2$ or $\exists y1(i-1), y2(i-1)$ not both. It is because the first elements of the array variables, $y1(1)$ and $y2(1)$, are initialized by $x1$ and $x2$, respectively, and the rest elements of the arrays, $y1(i), y2(i)$, for $2 \leq i \leq SIZE.y1$, are computed from their "ancestor" elements, $y1(i-1)$ and $y2(i-1)$. Scheduler of the MODEL compiler is able to detect such data dependency by examining array graphs [Lu81, MOD89]. It can statically decompose such a complex equation into the following simple equations:

```
Eq 1-1: y1(1) = IF x1 > x2 THEN x1 ELSE x2;
Eq 1-2: y1(i) = IF y1(i-1) > y2(i-1) THEN y1(i-1) - y2(i-1)
                                       ELSE y1(i-1);
```

[1]presented in Figure 2.1 of Chapter 2

`Eq 1-1` is similar to an initialization statement of an iteration block while `Eq 1-2` can be regarded as a body of the iteration block. Then the existence conditions for `Eq 1-1` is $\exists x1, x2$. On the other hand, the equation, `Eq 1-2`, has the existence condition such that $\forall i, 2 \leq i \leq SIZE.y1, \exists y1(i-1), y2(i-1)$.

We translate a MODEL equation into a CLIPS rule that performs the same function as the equation does. A CLIPS rule, in general, has two components: preconditions and actions. Whenever the preconditions of the rule are satisfied, the actions are executed. The CLIPS expert system maintains a list of facts stored in its knowledge base, called fact-list in CLIPS. The satisfiability of the preconditions is tested through pattern matching [GR89]. We propose that the existence conditions of a MODEL equation are translated into preconditions of a CLIPS rule; The body of the equation is denoted by the actions of a CLIPS rule; Every element of array variables in MODEL is represented by a CLIPS fact in the fact-list; A MODEL subscript is expressed as a variable in CLIPS.

A set of MODEL equations are translated into a collection of CLIPS rules. Input values of the equations are provided as facts for CLIPS. Then, the expert system performs the pattern matching with the patterns specified in the precondition part and the facts (the input values for the equations) stored in the fact-list. It checks if the facts of such patterns are in the fact-list (if the required inputs are available). If so, the values of the variables specified in the precondition part are obtained from the facts. It follows that the rules, whose preconditions are satisfied, are invoked. A set of new facts (the outputs of the equations) are generated and the fact-list is updated.

A CLIPS rule, `rule1`, which will be shown to be equivalent to the equation, `Eq 1-1`, is defined as follows:

```
(defrule rule1      ; for Eq 1-1
;  ''preconditions''
   (x1 ?x1)         ; get initial values of x1 and x2
   (x2 ?x2)
```

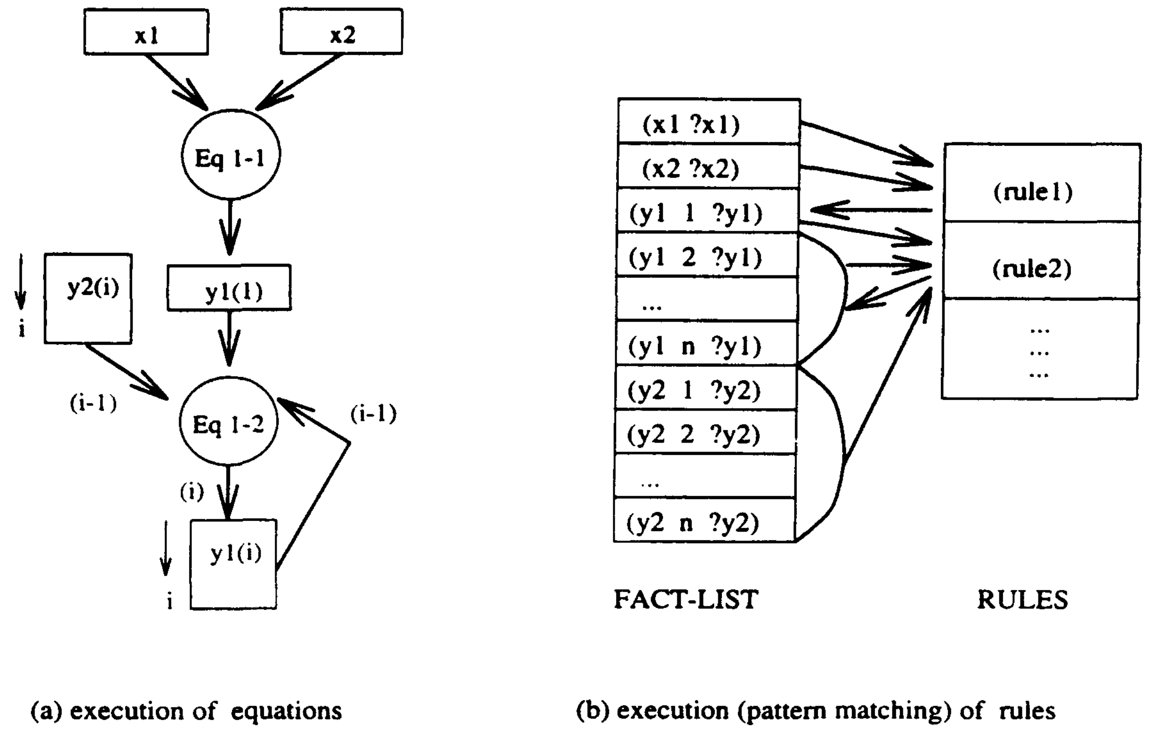


Figure 7.1: MODEL equations and their translation into CLIPS rules.

```
   =>
;  ''actions''
   (if (> ?x1 ?x2) then
             (assert (y1 1 ?x1))
                   else
             (assert (y1 1 ?x2))))
```

The variables, $x1$ and $x2$, are represented as CLIPS facts, `(x1 ?x1)` and `(x2 ?x2)` in Figure 7.1-(b), where **x1** and **x2** are relation names of the facts and ?x1 and ?x2 are CLIPS variables denoting their values. The CLIPS variables denote the values of the input variables of the equation, $x1$ and $x2$. The existence conditions of the equation, $\exists x1, x2$, are encoded as the preconditions of the rule. The body of the equation is translated into the actions of the rule. As shown in Figure 7.1, **rule1** defines a new fact, `(y1 1 ?y1)`, where ?y1 is a CLIPS variable for a value computed by the actions of the rule, from the input facts, `(x1 ?x1)` and `(x2 ?x2)`. It is equivalent to the computation such that the equation, Eq 1-1 defines the value of $y1(1)$ from $x1$ and $x2$. For example, the equation computes $y1(1) = 65$ if $x1 = 26$ and $x2 = 65$. The execution can be simulated by CLIPS as follows: The input values are provided by commands, (assert (x1 26)) and (assert (x1 65)). The satisfiability of the preconditions is checked through the pattern matching and the CLIPS variables, ?x1 and ?x2, get the values, 26 and 65, respectively. Since the preconditions are satisfied, the actions are executed. In this particular example, CLIPS command, (assert (y1 1 ?x2)), is invoked. Since x2 is asserted as 65, a new fact, (y1 1 65), is formulated and stored in the fact-list, as shown in Figure 7.1-(b).

Eq 1-2 is translated into the following rule, rule2:

```
(defrule rule2    ; for Eq 1-2
   (test (> ?i 1))       ; subscript i must be greater than 1.
   (y1 =(- ?i 1) ?y1)    ; y1 existence condition, namely,
                         ; the value of y1(i-1) exists.
   (y2 =(- ?i 1) ?y2)    ; y2 existence condition, namely,
                         ; the value of y2(i-1) exists.
   =>
   (if (> ?y1 ?y2) then
                (assert (y1 ?i =(- ?y1 ?y2)))
                else
                (assert (y1 ?i ?y1))))
```

The array variables, $y1(i)$ and $y2(i)$, of the equation are represented by multiple argument facts like (y1 i ?y1) and (y2 i ?y2), $1 \leq i \leq n$, as shown in Figure 7.1-(b). y1 and y2 denote relation names of the facts. The first argument, i, represents the subscript of the array variables (the index of the facts). The second arguments, ?y1 and ?y2, denote the values of the elements of the array variables, $y1(i)$ and $y2(i)$, respectively. The existence conditions of Eq 1-2 contains $\forall i, 2 \leq i \leq SIZE.y1$, where $SIZE.y1$ denotes the maximum value of the subscript i. (y1 =(- ?i 1)

`?y1)` means that there must exist $y1(i-1)$. If so, the expert system assigns a value to the variable, `?y1`. Similarly, `(y2 =(- ?i 1) ?y2)` checks the existence condition of $\exists y2(i-1)$. The actions of the rule represent the consequence of executing the equation. New facts, `(y1 2 ?y1)`, ..., `(y1 n ?y1)`, where `n` $= SIZE.y1$, are generated by the rule and they represent the elements of the array variable, $y1(2), ..., y1(n)$.

We can find the following similarities of the two systems:

- The variables of the equation can be represented by the facts of the expert system: The value of a MODEL variable cannot be changed after it is "assigned" (using "=") by an equation. Similarly, the fact of the expert system cannot be modified either once it is "asserted" (using the **assert** command) by a rule.

- The existence condition of the equation can be regarded as the preconditions of the rule: The equation is fired only when all of its inputs are available (the existence condition is true). Similarly, the rule is executed only when all of the preconditions, which check the existence of facts denoting the MODEL input variables, are satisfied.

- The consequences of executing the equation can be simulated by the actions of the rule: The assertion of new facts in the expert system is equivalent to the assignment of values to the outputs in MODEL.

It concludes that the execution of a MODEL equation is similar to the execution of a rule in the expert system, CLIPS. Therefore, an array graph for a MODEL specification can be translated into the sequence of pattern matching scheduled by the Rete algorithm [For82, GR89]. In many rule-based expert systems, the Rete algorithm is used in scheduling efficient pattern matching in a large collection of rules and facts. Normally, the fact-list of expert systems are modified during each cycle of the pattern matching process [GR89]. And the changes of the fact-list during each cycle are typically small percentage of the whole fact-list. Therefore we can reduce unnecessary computations of searching for facts by having the new or updated facts search for rules instead of having rules search for the whole facts in the fact-list. The details of the algorithm is described in [For82, GR89].

7.3 Translation

A MODEL variable is represented by a CLIPS fact: A subscript and a scalar are expressed by single argument facts. An array is denoted by a multiple argument fact. A MODEL equation is translated into CLIPS rule(s) depending on its existence conditions. The procedure of the translation is described in this section. An example of translation will be given in the next section.

The translation is performed in the following steps:

- Data declaration is translated: A subscript and a scalar variable are translated into single argument facts. A multi-dimensional array is converted to a multiple argument fact.
- The existence conditions of each equation are examined. Depending on the structure of the existence conditions, an equation may have to be decomposed. The scheduler of the MODEL compiler can statically decompose such a complex equation into simple ones each of which can be represented by a single CLIPS rule.
- The existence conditions of each equation are translated into the preconditions of the corresponding rule.
- The equation body of each equation is translated into the actions of the corresponding rule.

A MODEL variable is stored as a CLIPS fact. A scalar variable, `x1`, is represented by a CLIPS fact. Thus, a MODEL equation such as `x1 = 26;` is translated into an assertion of a new fact, `(assert (x1 26))`. On the other hand, the value can retrieved by the following command, `(x1 ?x1)`, where `x1` is the relation name and

`?x1` is a variable. Through the pattern matching between the asserted fact, `(x1 26)` and the command, `(x1 ?x1)`, the variable has the value, 26. A subscript variable is also denoted by a single argument fact. An element of an n-dimensional array such as $\mathtt{w}(\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_n)$, where $\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_n$ are subscripts, is expressed as a CLIPS fact such as `(w` i_1 i_2 ... i_n `val)`, where `val` is the value of the element. Its definition and retrieval are as same as those of the scalar variable.

A MODEL function is translated into a CLIPS function using the `deffunction` command.

A MODEL equation consists of two parts: the implicit existence condition and the equation body. The existence condition is translated into the preconditions of a CLIPS rule. The equation body becomes the actions of the rule.

7.4 Example

The equations of the gcd example shown in Figure 2.1, Chapter 2, are translated into CLIPS rules to illustrate the translation procedure.

As illustrated in Section 7.2, equation, `Eq 1`, has to be decomposed into two simple ones, `Eq 1-1` and `Eq 1-2`, due to its complex existence condition. The simple equations are translated into CLIPS rules, `rule1` and `rule2`, respectively. Similarly, equation, `Eq 2`:

```
Eq 2: y2(i) = IF i=1 THEN IF x1 > x2 THEN x2 ELSE x1
                      ELSE IF y1(i-1) > y2(i-1) THEN y2(i-1)
                                                ELSE y2(i-1) - y1(i-1);
```

should also be decomposed into the following two simple equations:

```
Eq 2-1: y2(1) = IF x1 > x2 THEN x2 ELSE x1;
Eq 2-2: y2(i) = IF y1(i-1) > y2(i-1) THEN y2(i-1)
                                      ELSE y2(i-1) - y1(i-1);
```

The existence conditions for Eq 2-1 is $\exists x1, x2$. The equation is translated into the following rule:

```
(defrule rule3        ; for Eq 2-1
   (x1 ?x1)           ; get initial values of x1 and x2
   (x2 ?x2)
   =>
   (if (> ?x1 ?x2) then
             (assert (y2 1 ?x2))
                 else
             (assert (y2 1 ?x1))))
```

The equation, Eq 2-2, has the following existence condition:

$$\forall i, 2 \leq i \leq SIZE.y2, \exists y1(i-1), y2(i-1)$$

The rules are translated into the following rules:

```
(defrule rule4        ; for Eq 2-2
   (test (> ?i 1))        ; subscript i must be greater than 1.
   (y1 =(- ?i 1) ?y1)     ; y1 existence condition, namely,
                          ; the value of y1(i-1) exists.
   (y2 =(- ?i 1) ?y2)     ; y2 existence condition, namely,
                          ; the value of y2(i-1) exists.
   =>
   (if (> ?y1 ?y2) then
                 (assert (y2 ?i ?y2))
                 else
                 (assert (y2 ?i =(- ?y2 ?y1)))))
```

Those four rules, `rule1` to `rule4`, compute the facts, $\forall i, 1 \leq i \leq SIZE.y1$, `(y1 i y1-i)`, `(y2 i y2-i)`, where `y1-i` and `y2-i` represent the values of `y1(i)` and `y2(i)`, respectively.

The equation, Eq 3:

```
Eq 3: END.y1(i) = (y1(i) = y2(i));
```

becomes the following rule:

```
(defrule rule5
   (y1 ?i ?y1) ; y1 existence condition
   (y2 ?i ?y2) ; y2 existence condition
   =>
   (assert (end-y1 ?i =(= ?y1 ?y2))))
```

The multiple argument fact, `end-y1)`, contains boolean values in its second argument. The boolean value is determined by evaluating expression, `=(= ?y1 ?y2)`. The expression returns *true* if the values of `?y1` and `?y2`, which represent the MODEL variables, $y1(i)$ and $y2(i)$, respectively, are same. Otherwise, it returns *false*. The first argument, `?i`, is an index of the fact which is in fact a subscript of a MODEL variable, $END.y1$. Since $y1(i) \neq y2(i)$ for all $i, 1 \leq i \leq SIZE.y1$, the fact looks like as follows: `(end-y1 1 false)`, `(end-y1 2 false)`,..., `(end-y1 SIZE.y1 - 1 false)`, `(end-y1 SIZE.y1 true)`.

Finally, the equation, Eq 4:

```
Eq 4: z = IF (END.y1(i)) THEN y1(i);
```

becomes the following rule:

```
(defrule rule6
   (y1 ?i ?y1)          ; y1 existence condition
   (end-y1 ?i ?end-y1) ; end-y1 existence condition
   =>
   (if (?end-y1) then
        (assert (z ?y1))))
```

Chapter 8

Conclusion

8.1 Summary

The concept of a visual software environment has been investigated. The environment facilitates man-machine cooperation, especially the "oracle" operations, during software development. The environment supports visual programming, compilation. testing, verification, and knowledge acquisition. The environment is designed exclusively for the equational language, MODEL, and implemented using DECdesign.

The visual programming is exercised using an icon-based graph editor. The user draws an array graph and keys-in the equations and declarations to define nodes of the graph precisely. The array graph visualizes an equational specification as follows: The nodes of the graph represent equations and variables of the equational specification. The edges of the graph denote hierarchical, data, and parameter dependencies among the equations and the variables. The graph is displayed in a graphics window. This helps a user in perceiving the equational specification. The mathematical definitions of the variables and the equations in MODEL are displayed in a text window. The combination of graphics and equations facilitates software understanding. The understanding is further facilitated by the compilation, testing, and verification capabilities.

The syntax analysis of the array graph is performed interactively while a user composes the graph. The MODEL compiler examines the array graph to check if (1) there are ambiguous or incomplete definitions of variables and equations, (2) variables references are consistent, (3) there is a causality chain that computes a solution set for a given input values, and (4) conditions of terminating programs are specified. The interactive syntax analysis and the compilation detect errors at the early stage of program development. Messages from the syntax analysis and the checking are visually expressed in the array graph.

The Equational Visual Testing is based on the array graph that directly visualizes data flow of an equational specification. (Note that a control flow graph used in the Procedural Testing does not directly express such data flow information.) The test adequacy criteria of the Equational Visual Testing are defined in terms of visualized paths of the array graph. The adequacy criteria are: (1) The equational all-paths criterion defines a finite number of source-to-target paths based on acyclic array graphs (the equational compiler automatically opens an MSCC of the array graph). This makes the criterion practical. (Note that the procedural all-paths criterion requires traversal of undefined number of paths.) (2) The equational all-du-paths criterion defines at most a two-edge path, from an equation node, where a variable is defined, to equation nodes, where the variable is used. The path can be selected for traversal by providing test input values that satisfy at most two conditions. (Note that the procedural all du-paths criterion may require finding test input values that satisfy a conjunction of more than two conditions.) (3) The equational all-uses criterion is based on a single-edge path from a variable node to an equation node where the variable is used. The path can be traversed by satisfying at most one condition. This reduces labor of the testing in length of conjunction of the conditions. (4) The equational all-definitions criterion is shown to be trivial. Any single execution will traverse all definition paths. (Note that the procedural all-definitions criterion requires to traverse multiple assignment statements which define and update variables.)

The visual software environment facilitates performing the "oracle" operations in the testing as follows. The conjunctions of the conditions are listed in a condition table. The rows of the condition table are marked after test execution, according to the test results. The condition nodes of the array graph are shaded when they are satisfied during test execution. The number of the satisfied rows in the condition table denotes the progress of the testing. The backtracking along a causality path from a specific condition to the respective input variables in the array graph is facilitated. This reduces the labor of finding the test input values. Each path can be traversed only if the conjunction of all conditions on the path is satisfied. If any of the condition or the conjunction of the conditions is unsatisfiable, the path is not feasible. The human tester can view interim values of the variables and also refer to a specific iteration. This is possible because every element of array variables is defined and the interim values are recorded during test execution. This further reduces the labor of evaluating test results.

The verification employs equational reasoning. There is no need to trace program states during the program verification because of the single assignment rule and the referential transparency. Note that program verification systems for the procedural programming are complicated because the systems trace transitions in program states. The basic deduction rules of the equational reasoning system are Reflexivity, Symmetry, Transitivity, Replacement and Substitutivity. The user deduces equations and applies general algebraic laws using deduction rules.

There is a great deal of valuable expertise in old legacy programs. They can be automatically translated to equations that are essentially rules in rule-based expert systems. The environment allows users to extract such expertise from old legacy programs and accumulate it as rules in knowledge bases. This is based on language translations from a procedural language to MODEL and from MODEL to a rule-based language. The tools of automatic language translation reduce human labor in collecting expertise for a rule-based expert system.

8.2 Future Research

The environment can be further extended in many ways. The followings are examples:

(1) **Empirical study of comparing the adequacy criteria**

The strictness of the testing is determined by its adequacy criterion. The stronger the criterion is, the more errors can be found. But the stronger criterion requires more tests. An empirical study can be made to compare the equational adequacy criteria in terms of the required number of tests and the number of errors found. The result will be helpful for the tester in his planning the testing.

(2) **Symbolic manipulation for input data selection**

Finding test input values which satisfy a certain condition is an oracle operation. We can find symbolic relations between expressions that impose constraints of input variables. This is facilitated by backtracking along causality paths in the array graph. A symbolic manipulator and simplifier would reduce the labor of finding test input values.

Appendix A

MODEL Grammar

```
<equation> ::= <cond_eqn>
     | <simple_eqn>

<cond_eqn> ::= <mvar> = <cond_exp> ;

<simple_eqn> ::= <mvar> = <bool_exp> ;

<bool_exp> ::= <cond_exp>
             | <bool_term_s>

<bool_term_s> ::= <bool_term>
                | <bool_term_s> OR <bool_term>

<cond_exp> ::= IF <bool_term_s> THEN <bool_exp>
             | IF <bool_term_s> THEN <bool_exp> ELSE <bool_exp>

<bool_exp_s> ::= <bool_exp>
       | <bool_exp_s> , <bool_exp>
```

```
<bool_term> ::= <bool_factor>
      | <bool_term> AND <bool_factor>

<bool_factor> ::= <arith_exp>
         | <bool_factor> GE <arith_exp>
         | <bool_factor> LE <arith_exp>
         | <bool_factor> NE <arith_exp>
         | <bool_factor> NG <arith_exp>
         | <bool_factor> NL <arith_exp>
         | <bool_factor> GT <arith_exp>
         | <bool_factor> LT <arith_exp>
         | <bool_factor> = <arith_exp>

<arith_exp> ::= <term>
      | <arith_exp> + <term>
      | <arith_exp> - <term>

<term> ::= <factor>
         | <term> * <factor>
         | <term> / <factor>

<factor> ::= <primary>
   | - <primary>
   | ^ <primary>
   | <factor> EXPO <primary>

<primary> ::= <const>
    | <sub_var>
    | ( <bool_exp> )
```

```
<const> ::= NUM
          | CHAR

<mvar> ::= <sub_var>

<sub_var> ::= <varble>
    | <varble> ( <bool_exp_s> )

<varble> ::= VAR
```

Appendix B

Array Graph Grammar

```
<array graph> ::= <source list> <equations> <target list>;
                  { if ($1.fields != $2.in_flds ||
                        $2.out_flds != $3.fields) then PARSING_ERROR }

<source list> ::= <source list> <source list>
                  { $0.fields = UNION ($1.fields, $2.fields) }
                | ( <source list> )
                  { $0.fields = $2.fields }
                | <file sym> ( <in rec list> )
                  { $0.fields = $3.fields }
                | NULL
                  { $0.fields = NULL }

<in rec list> ::= <in rec list> <in rec list>
                    { $0.fields = UNION ($1.fields, $2.fields) }
                  | <h conn> <rec sym> ( <in grp list> )
                    { $0.fields = $4.fields }
                  | <h conn> <rec sym> ( <in fld list> )
                    { $0.fields = $4.fields }

<in grp list> ::= <in grp list> <in grp list>
```

```
                    { $0.fields = UNION ($1.fields, $2.fields) }
                  | <h conn> <grp sym> ( <in rec list> )
                    { $0.fields = $4.fields }
                  | <h conn> <grp sym> ( <in grp list> )
                    { $0.fields = $4.fields }
                  | <h conn> <grp sym> ( <in fld list> )
                    { $0.fields = $4.fields }

<in fld list> ::= <in fld list> <in fld list>
                    { $0.fields = UNION ($1.fields, $2.fields) }
                  | <h conn> <fld sym>
                    { $0.fields = UNION (NULL, [$2.name,$2.data-type]) }

<target list> ::= <target list> <target list>
                    { $0.fields = UNION ($1.fields, $2.fields) }
                  | ( <target list> )
                    { $0.fields = $2.fields }
                  | ( <out rec list> ) <file sym>
                    { $0.fields = $2.fields }

<out rec list> ::= <out rec list> <out rec list>
                     { $0.fields = UNION ($1.fields, $2.fields) }
                   | ( <out grp list> ) <rec sym> <h conn>
                     { $0.fields = $2.fields }
                   | ( <out fld list> ) <rec sym> <h conn>
                     { $0.fields = $2.fields }

<out grp list> ::= <out grp list> <out grp list>
                     { $0.fields = UNION ($1.fields, $2.fields) }
                   | ( <out rec list> ) <grp sym> <h conn>
                     { $0.fields = $2.fields }
                   | ( <out grp list> ) <grp sym> <h conn>
                     { $0.fields = $2.fields }
                    | ( <out fld list> ) <grp sym> <h conn>
                      { $0.fields = $2.fields }

<out fld list> ::= <out fld list> <out fld list>
                      { $0.fields = UNION ($1.fields, $2.fields) }
                    |<fld sym> <h conn>
                      { $0.fields = UNION (NULL, [$1.name,$1.data-type]) }

<equations> ::= <equations> <equations>
                  { $0.in_flds = UNION ($1.in_flds,
                                        $2.in_flds - $1.out_flds)
                    $0.out_flds = UNION ($1.out_flds - $2.in_flds,
                                         $2.out_flds) }
                | ( <equations> )
                  { $0.in_flds = $2.in_flds
                    $0.out_flds = $2.out_flds }
                | ( <eq input list> ) <eq sym> <eq output>
                  { $0.in_flds = $2.in_flds
                    $0.out_flds = UNION (NULL, [$5.name,$5.data-type]) }

<eq input list> ::= <eq input list> <eq input list>
                      { $0.in_flds = UNION ($1.in_flds, $2.in_flds) }
                    | <fld sym> <d conn>
                      { $0.in_flds = UNION (NULL, [$1.name,$1.data-type]) }

<eq output> ::= <d conn> <fld sym>
                { $0.out_flds = UNION (NULL, [$2.name,$2.data-type]) }
```

Appendix C

MODEL Syntax Error Messages

Message Id.	Message
1	BIT STRING CONTAINS CHARACTER OTHER THAN 0 OR 1
2	COLON MISSING AFTER THE WORD "BLOCK"
3	BADLY FORMED BOOLEAN EXPRESSION AFTER IF IN-STATEMENT
4	MISSING OR INVALID NUMERIC CONSTANT IN ITERATIVE COUNT SPEC
5	MISSING OR INVALID NUMERIC CONSTANT IN RELATIVE ERROR SPEC
7	ORGANIZATION TYPE MISSING OR ILLEGAL IN DISK STATEMENT
9	TYPE DISK MISSING OR ILLEGAL IN DISK STATEMENT
12	MISSING ELSE IN CONDITIONAL EXPRESSION
14	ASSERTION MISSING AFTER THE KEYWORD "THEN"
18	NO BOOLEAN EXPRESSION AFTER THE KEYWORD "IF"
22	NO EXPRESSION AFTER LEFT PARENTHESIS
23	KEYWORD "=" IS MISSING
24	RIGHT PARENTHESIS MISSING
26	STRING MISSING AFTER QUOTE
33	ERROR IN RECOGNITION OF RIGHT HAND SIDE OF AN ASSERTION

Message Id.	Message
38	KEYWORD "THEN" IS MISSING
39	RECORD OR GROUP KEYWORD EXPECTED
42	RECORD NAME MISSING OR ILLEGAL IN FILE OR REPORT STATEMENT
44	MEDIUM NAME MISSING OR ILLEGAL IN FILE OR REPORT
45	KEYNAME MISSING IN FILE OR REPORT STATEMENT
46	MAXIMUM LENGTH MISSING OR ILLEGAL IN VARIABLE LENGTH IN FIELD STATEMENT
47	INVALID OR MISSING FIELD TYPE IN FIELD/INTERIM STATEMENT
48	MISSING OR INVALID LENGTH IN FIELD/INTERIM STATEMENT
49	MISSING RIGHT PARENTHESIS AFTER FIELD-TYPE IN FIELD/INTERIM
50	MINUS SIGN IS NOT FOLLOWED BY AN INTEGER
51	MISSING/INVALID MAX NUMBER OF OCCURRENCES OF ITEMS.
52	NAME MISSING OR ILLEGAL IN ITEM LIST
53	MISSING LEFT PARENTHESIS IN LINE SPEC
54	MISSING INTEGER IN LINE SPEC
55	MISSING RIGHT PARENTHESIS IN LINE SPEC
56	MISSING/INVALID FILE NAME AFTER KEYWORD FILE
57	FORMAT MISSING/MISSPELLED AFTER RECORD IN STORAGE STATEMENT
58	MISSING/INVALID TAPE LABEL
59	KEYWORD "RECORDSIZE" MISSING OR MISSPELLED AFTER "MAX"
60	MISSING/INVALID VOLUME NAME (EXTERNAL OR INTERNAL)
61	MISSING/INVALID DEVICE TYPE
62	MISSING/INVALID ITERATIVE SOLUTION METHOD
63	COLON MISSING AFTER KEYWORD "MODULE"

Message Id.	Message
64	NAME MISSING OR ILLEGAL IN MODULE STATEMENT
65	ERROR IN ASSEMBLY OF A NUMBER CONSTANT
66	TAPE SPEC PARAMETER MISSING OR ILLEGAL
67	ERROR IN PICTURE SPEC
68	QUALIFIED NAME ILLEGAL
69	RECORD FORMAT MISSING OR ILLEGAL
70	KEYWORD "BLOCKSIZE" MISSING IN RECORD FORMAT SPEC
71	BLOCKSIZE VALUE MISSING/ILLEGAL IN RECORD FORMAT SPEC
72	RECORD SIZE VALUE MISSING/ILLEGAL IN RECORD FORMAT SPEC
74	SEMICOLON MISSING AT END OF STATEMENT
75	COLON MISSING AFTER KEYWORD "SOURCE"
76	NAME MISSING/ILLEGAL IN SOURCE FILE LIST
77	COLON MISSING AFTER KEYWORD "TARGET"
78	NAME MISSING/ILLEGAL IN TARGET FILE LIST
79	MISSING "THEN" IN CONDITIONAL EXPRESSION
80	UNRECOGNIZABLE STATEMENT
81	BADLY FORMED ARITHMETIC EXPRESSION
82	BADLY FORMED BOOLEAN EXPRESSION
83	BADLY FORMED BOOLEAN TERM
84	BADLY FORMED CONCATENATION OF EXPRESSIONS
85	BADLY FORMED FACTOR
86	BADLY FORMED PRIMARY
87	BADLY FORMED TERM

Message Id.	Message
90	LEFT PARENTHESIS MISSING IN COLUMN SPEC
91	INTEGER MISSING IN COLUMN SPEC
92	RIGHT PARENTHESIS MISSING IN COLUMN SPEC
101	LENGTH OF PICTURE SPECIFICATION IS TOO SMALL
102	SPECIFIED LENGTH IS INAPPROPRIATE FOR SPECIFIED TYPE OF DATA
104	SPECIFIED MAXIMUM LENGTH IS INAPPROPRIATE OR TOO SMALL
105	FRACTION POINT OFFSET IS OUTSIDE OF BOUNDS $-128 < P < 127$
106	BAD REPETITION SPECIFICATION
107	ILLEGAL CHARACTER IN PICTURE SPECIFICATION
108	EXPECTING A LEVEL NUMBER IN A STRUCTURED DATA DESCRIPTION STATEMENT
109	LENGTH OF PICTURE SPECIFICATION IS TOO BIG
110	ILLEGAL BIT STRING IN "ON_CERR" CLAUSE
111	INCONSISTENT USE OF "ON_CERR" CLAUSE AND THE ATTRIBUTE OF THE FIELD
112	INVALID SPECIFICATION IN "ON_CERR" CLAUSE
113	ILLEGAL B2 CONSTANT
114	ILLEGAL B3 CONSTANT
115	ILLEGAL B4 CONSTANT
120	MORE THEN ONE SOURCE FILE IN FUNCTION SPECIFICATION
121	MORE THEN ONE TARGET FILE IN FUNCTION SPECIFICATION
122	MORE THEN ONE RECORD IN FUNCTION FILE DEFINITION
123	GROUPS ARE NOT ALLOWED IN FUNCTION FILE DEFINITIONS

Message Id.	Message
150	SUBLINEAR FUNCTION TAKES TWO PARAMETERS, BOTH CONDITIONAL EXPRESSIONS
151	SUBLINEAR FUNCTION ONLY TAKES TWO PARAMETERS, BOTH CONDITIONAL EXPRESSIONS
153	SUBLINEAR FUNCTION TAKES TWO PARAMETERS, BOTH CONDITIONAL EXPRESSIONS
160	MISSING COLON AFTER KEYWORD "PROCEDURE"
161	MISSING/INVALID SPECIFICATION NAME
162	MISSING/INVALID PARAMETER NAME
163	MISSING/INVALID EXTERNAL NAME
164	MISSING/INVALID I/O MODE SPECIFICATION (MUST BE "IN", "OUT", OR "INOUT")
165	MISSING "." AFTER PACKAGE NAME
166	MISSING/INVALID PACKAGE NAME
168	MISSING/INVALID TYPE NAME
169	MISSING/INVALID PACKAGE NAME AFTER RENAME
170	MISSING/INVALID NAME FOR RENAMING

[DEC91] Digital Equipment Corporation, Maynard, Massachusetts. *DECdesign User's Guide*, May 1991.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[DMMP87] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings Publishing Co., Menlo Park, California, 1987.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, Potomac, Maryland, 1979.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Ge89] Xiang Ge. *An Intelligent Mathematical Modeling System*. PhD thesis, University of Pennsylvania, 1989.

[GJ90] D. Gelernter and S. Jagannathan. *Programming Linguistics*. The MIT Press, Cambridge, Massachusetts, 1990.

[GP89] Xiang Ge and Noah S. Prywes. Reverse software engineering of concurrent programs. manuscript, 1989.

[GR89] J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. PWS-KENT Publishing Company, 1989.

[Ham88] R. Hamlet. Special section on software testing. *Communications of the ACM*, 31(6):662–667, June 1988.

[How87] W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill Inc., New York, New York, 1987.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hud91] Paul Hudak. Para-functional programming in Haskell. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–158. ACM Press, 1991.

[Kim91] Jee-In Kim. Equational and rule-based programming: Visualization, reliability, and knowledge base generation. Technical Report MS-CIS-91-60, Department of Computer and Information Science, University of Pennsylvania, 1991.

[Kro87] F. Kroger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, New York, New York, 1987.

[Lam83] L. Lamport. What good is temporal logic? In *Proceedings IFIP Congress, Paris*, pages 657–668, Amsterdam, Holland, 1983. North-Holland.

[Lin88] P. A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, pages 3–27, January 1988.

[LP90] Evan Lock and Noah S. Prywes. Software engineering environment for parallel/concurrent programs on a computer network. Technical report, Computer Command and Control Company, 2300 Chestnut Street, Philadelphia, PA 19103, 1990.

[Lu81] K. S. Lu. *Program Optimization Based on a Non-Procedural Specification.* PhD thesis, University of Pennsylvania, 1981.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Computer Science Series. McGraw-Hill Book Co., New York, New York, 1974.

[MOD89] Computer Command and Control Company, 2300 Chestnut Street, Philadelphia, PA 19103. *The MODEL Compiler Usage and Reference Guide — Non-Procedural Programming for Non-Programmers*, 1989.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, England, 1981.

[Nta84] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[PGLS90] Noah S. Prywes, Xiang Ge, Insup Lee, and Mitchel Song. Procedural to equational language translation. Technical Report Contract AFSOR-88-0116, Department of Computer and Information Science, University of Pennsylvania, 1990.

[PLGS88] Noah S. Prywes, Insup Lee, Xiang Ge, and Mitchel Song. Reverse software engineering. Technical Report MS-CIS-88-99, Department of Computer and Information Science, University of Pennsylvania, 1988.

[PLK91] Noah S. Prywes, Insup Lee, and Jee-In Kim. Extracting rules from software for knowledge bases. In *Proceedings of Intelligent Information System Workshop at Rome, NY.*, October 1991.

[PP83] Noah S. Prywes and A. Pnueli. Compilation of nonprocedural specifications into computer programs. *IEEE Transactions on Software Engineering*, SE-9(3):267–279, May 1983.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[SLPP84] B. Szymanski, Evan Lock, A. Pnueli, and Noah S. Prywes. On the scope of static checking in definitional languages. In *Proceedings of the ACM Annual Conference*, pages 197–207, San Francisco, California, October 1984.

[SP88] B. K. Szymanski and Noah S. Prywes. Efficient handling of data structures in definitional languages. *Science of Computer Programming*, 10:221–245, 1988.

[Szy91] B. K. Szymanski. EPL — parallel programming with recurrent equations. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 51–104. ACM Press, 1991.

[Wey86] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

[Wey90] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, SE-16(2):121–128, February 1990.